root, however, and in this part of the question, we will ask how sensitive a multiple root can be: First, write $p(x) = q(x) \cdot (x - \mathrm{r(i)})^m$, where $q(\mathrm{r(i)}) \neq 0$ and m is the multiplicity of the root r(i). Then compute the m roots nearest r(i) of the slightly perturbed polynomial $p(x) - q(x)\epsilon$, and show that they differ from r(i) by $|\epsilon|^{1/m}$. So that if $m = 2$, for instance, the root r(i) is perturbed by $\epsilon^{1/2}$, which is much larger than ϵ if $|\epsilon| \ll 1$. Higher values of m yield even larger perturbations. If ϵ is around machine epsilon and represents rounding errors in computing the root, this means an m-tuple root can lose all but $1/m$-th of its significant digits.

QUESTION 1.21. *(Medium)* Apply Algorithm 1.1, Bisection, to find the roots of $p(x) = (x - 2)^9 = 0$, where $p(x)$ is evaluated using Horner's rule. Use the Matlab implementation in HOMEPAGE/Matlab/bisect.m, or else write your own. Confirm that changing the input interval slightly changes the computed root drastically. Modify the algorithm to use the error bound discussed in the text to stop bisecting when the roundoff error in the computed value of $p(x)$ gets so large that its sign cannot be determined.

2

Linear Equation Solving

2.1. Introduction

This chapter discusses perturbation theory, algorithms, and error analysis for solving the linear equation $Ax = b$. The algorithms are all variations on Gaussian elimination. They are called *direct methods*, because in the absence of roundoff error they would give the exact solution of $Ax = b$ after a finite number of steps. In contrast, Chapter 6 discusses *iterative methods*, which compute a sequence $x_0, x_1, x_2, \ldots$ of ever better approximate solutions of $Ax = b$; one stops iterating (computing the next x_{i+1}) when x_i is accurate enough. Depending on the matrix A and the speed with which x_i converges to $x = A^{-1}b$, a direct method or an iterative method may be faster or more accurate. We will discuss the relative merits of direct and iterative methods at length in Chapter 6. For now, we will just say that direct methods are the methods of choice when the user has no special knowledge about the source[7] of matrix A or when a solution is required with guaranteed stability and in a guaranteed amount of time.

The rest of this chapter is organized as follows. Section 2.2 discusses perturbation theory for $Ax = b$; it forms the basis for the practical error bounds in section 2.4. Section 2.3 derives the Gaussian elimination algorithm for dense matrices. Section 2.4 analyzes the errors in Gaussian elimination and presents practical error bounds. Section 2.5 shows how to improve the accuracy of a solution computed by Gaussian elimination, using a simple and inexpensive iterative method. To get high speed from Gaussian elimination and other linear algebra algorithms on contemporary computers, care must be taken to organize the computation to respect the computer memory organization; this is discussed in section 2.6. Finally, section 2.7 discusses faster variations of Gaussian elimination for matrices with special properties commonly arising in practice, such as symmetry ($A = A^T$) or sparsity (when many entries of A are zero).

[7] For example, in Chapter 6 we consider the case when A arises from approximating the solution to a particular differential equation, Poisson's equation.

Sections 2.2.1 and 2.5.1 discuss recent innovations upon which the software in the LAPACK library depends.

There are a variety of open problems, which we shall mention as we go along.

2.2. Perturbation Theory

Suppose $Ax = b$ and $(A + \delta A)\hat{x} = b + \delta b$; our goal is to bound the norm of $\delta x \equiv \hat{x} - x$. Later, $\hat{x}$ will be the computed solution of $Ax = B$. We simply subtract these two equalities and solve for δx: one way to do this is to take

$$\begin{array}{rcl} (A+\delta A)(x+\delta x) & = & b+\delta b \\ - \qquad [Ax & = & b] \\ \hline \delta Ax + (A+\delta A)\delta x & = & \delta b \end{array}$$

and rearrange to get

$$\delta x = A^{-1}(-\delta A\hat{x} + \delta b). \tag{2.1}$$

Taking norms and using part 1 of Lemma 1.7 as well as the triangle inequality for vector norms, we get

$$\|\delta x\| \le \|A^{-1}\|(\|\delta A\| \cdot \|\hat{x}\| + \|\delta b\|). \tag{2.2}$$

(We have assumed that the vector norm and matrix norm are consistent, as defined in section 1.7. For example, any vector norm and its induced matrix norm will do.) We can further rearrange this inequality to get

$$\frac{\|\delta x\|}{\|\hat{x}\|} \le \|A^{-1}\| \cdot \|A\| \cdot \left(\frac{\|\delta A\|}{\|A\|} + \frac{\|\delta b\|}{\|A\| \cdot \|\hat{x}\|}\right). \tag{2.3}$$

The quantity $\kappa(A) = \|A^{-1}\| \cdot \|A\|$ is the *condition number*[8] of the matrix A, because it measures the relative change $\frac{\|\delta x\|}{\|\hat{x}\|}$ in the answer as a multiple of the relative change $\frac{\|\delta A\|}{\|A\|}$ in the data. (To be rigorous, we need to show that inequality (2.2) is an equality for some nonzero choice of δA and δb; otherwise $\kappa(A)$ would only be an upper bound on the condition number. See Question 2.3.) The quantity multiplying $\kappa(A)$ will be small if δA and δb are small, yielding a small upper bound on the relative error $\frac{\|\delta x\|}{\|\hat{x}\|}$.

The upper bound depends on δx (via $\hat{x}$), which makes it seem hard to interpret, but it is actually quite useful in practice, since we know the computed solution $\hat{x}$ and so can straightforwardly evaluate the bound. We can also derive a theoretically more attractive bound that does not depend on δx as follows:

[8] More pedantically, it is the condition number with respect to the problem of matrix inversion. The problem of finding the eigenvalues of A, for example, has a different condition number.

LEMMA 2.1. *Let $\|\cdot\|$ satisfy $\|AB\| \le \|A\| \cdot \|B\|$. Then $\|X\| < 1$ implies that $I - X$ is invertible, $(I - X)^{-1} = \sum_{i=0}^{\infty} X^i$, and $\|(I - X)^{-1}\| \le \frac{1}{1-\|X\|}$.*

Proof. The sum $\sum_{i=0}^{\infty} X^i$ is said to converge if and only if it converges in each component. We use the fact (from applying Lemma 1.4 to Example 1.6) that for any norm, there is a constant c such that $|x_{jk}| \le c \cdot \|X\|$. We then get $|(X^i)_{jk}| \le c \cdot \|X^i\| \le c \cdot \|X\|^i$, so each component of $\sum X^i$ is dominated by a convergent geometric series $\sum c\|X\|^i = \frac{c}{1-\|X\|}$ and must converge. Therefore $S_n = \sum_{i=0}^{n} X^i$ converges to some S as $n \to \infty$, and $(I - X)S_n = (I - X)(I + X + X^2 + \cdots + X^n) = I - X^{n+1} \to I$ as $n \to \infty$, since $\|X^i\| \le \|X\|^i \to 0$. Therefore $(I - X)S = I$ and $S = (I - X)^{-1}$. The final bound is $\|(I - X)^{-1}\| = \|\sum_{i=0}^{\infty} X^i\| \le \sum_{i=0}^{\infty} \|X^i\| \le \sum_{i=0}^{\infty} \|X\|^i = \frac{1}{1-\|X\|}$. □

Solving our first equation $\delta Ax + (A + \delta A)\delta x = \delta b$ for δx yields

$$\begin{array}{rcl} \delta x & = & (A+\delta A)^{-1}(-\delta Ax + \delta b) \\ & = & [A(I + A^{-1}\delta A)]^{-1}(-\delta Ax + \delta b) \\ & = & (I + A^{-1}\delta A)^{-1}A^{-1}(-\delta Ax + \delta b). \end{array}$$

Taking norms, dividing both sides by $\|x\|$, using part 1 of Lemma 1.7 and the triangle inequality, and assuming that δA is small enough so that $\|A^{-1}\delta A\| \le \|A^{-1}\| \cdot \|\delta A\| < 1$, we get the desired bound:

$$\begin{array}{rcll} \frac{\|\delta x\|}{\|x\|} & \le & \|(I + A^{-1}\delta A)^{-1}\| \cdot \|A^{-1}\| \left(\|\delta A\| + \frac{\|\delta b\|}{\|x\|}\right) & \\ & \le & \frac{\|A^{-1}\|}{1 - \|A^{-1}\| \cdot \|\delta A\|} \left(\|\delta A\| + \frac{\|\delta b\|}{\|x\|}\right) & \text{by Lemma 2.1} \\ & = & \frac{\|A^{-1}\| \cdot \|A\|}{1 - \|A^{-1}\| \cdot \|A\| \frac{\|\delta A\|}{\|A\|}} \left(\frac{\|\delta A\|}{\|A\|} + \frac{\|\delta b\|}{\|A\| \cdot \|x\|}\right) & \\ & \le & \frac{\kappa(A)}{1 - \kappa(A)\frac{\|\delta A\|}{\|A\|}} \left(\frac{\|\delta A\|}{\|A\|} + \frac{\|\delta b\|}{\|b\|}\right) & (2.4) \\ & & \text{since } \|b\| = \|Ax\| \le \|A\| \cdot \|x\|. & \end{array}$$

This bound expresses the relative error $\frac{\|\delta x\|}{\|x\|}$ in the solution as a multiple of the relative errors $\frac{\|\delta A\|}{\|A\|}$ and $\frac{\|\delta b\|}{\|b\|}$ in the input. The multiplier, $\kappa(A)/(1 - \kappa(A)\frac{\|\delta A\|}{\|A\|})$, is close to the condition number $\kappa(A)$ if $\|\delta A\|$ is small enough.

The next theorem explains more about the assumption that $\|A^{-1}\| \cdot \|\delta A\| = \kappa(A) \cdot \frac{\|\delta A\|}{\|A\|} < 1$: it guarantees that $A + \delta A$ is nonsingular, which we need for δx to exist. It also establishes a geometric characterization of the condition number.

THEOREM 2.1. *Let A be nonsingular. Then*

$$\min\left\{\frac{\|\delta A\|_2}{\|A\|_2} : A + \delta A \text{ singular}\right\} = \frac{1}{\|A^{-1}\|_2 \cdot \|A\|_2} = \frac{1}{\kappa(A)}.$$

Therefore, the distance to the nearest singular matrix (ill-posed problem) $= \frac{1}{\text{condition number}}$.

Proof. It is enough to show $\min\{\|\delta A\|_2 : A+\delta A \text{ singular}\} = \frac{1}{\|A^{-1}\|_2}$.

To show this minimum is at least $\frac{1}{\|A^{-1}\|_2}$, note that if $\|\delta A\|_2 < \frac{1}{\|A^{-1}\|_2}$, then $1 > \|\delta A\|_2 \cdot \|A^{-1}\|_2 \geq \|A^{-1}\delta A\|_2$, so Lemma 2.1 implies that $I + A^{-1}\delta A$ is invertible, and so $A + \delta A$ is invertible.

To show the minimum equals $\frac{1}{\|A^{-1}\|_2}$, we construct a δA of norm $\frac{1}{\|A^{-1}\|_2}$ such that $A + \delta A$ is singular. Note that since $\|A^{-1}\|_2 = \max_{x\neq 0} \frac{\|A^{-1}x\|_2}{\|x\|_2}$, there exists an x such that $\|x\|_2 = 1$ and $\|A^{-1}\|_2 = \|A^{-1}x\|_2 > 0$. Now let $y = \frac{A^{-1}x}{\|A^{-1}x\|_2} = \frac{A^{-1}x}{\|A^{-1}\|_2}$ so $\|y\|_2 = 1$. Let $\delta A = \frac{-xy^T}{\|A^{-1}\|_2}$.

Then

$$\|\delta A\|_2 = \max_{z\neq 0} \frac{\|xy^Tz\|_2}{\|A^{-1}\|_2\,\|z\|_2} = \max_{z\neq 0} \frac{|y^Tz|}{\|z\|_2} \frac{\|x\|_2}{\|A^{-1}\|_2} = \frac{1}{\|A^{-1}\|_2},$$

where the maximum is attained when z is any nonzero multiple of y, and $A+\delta A$ is singular because

$$(A+\delta A)y = Ay - \frac{xy^Ty}{\|A^{-1}\|_2} = \frac{x}{\|A^{-1}\|_2} - \frac{x}{\|A^{-1}\|_2} = 0\,. \quad \square$$

We have now seen that the distance to the nearest ill-posed problem equals the reciprocal of the condition number for two problems: polynomial evaluation and linear equation solving. This reciprocal relationship is quite common in numerical analysis [71].

Here is a slightly different way to do perturbation theory for $Ax = b$; we will need it to derive practical error bounds later in section 2.4.4. If $\hat{x}$ is any vector, we can bound the difference $\delta x \equiv \hat{x} - x = \hat{x} - A^{-1}b$ as follows. We let $r = A\hat{x} - b$ be the *residual* of $\hat{x}$; the residual r is zero if $\hat{x} = x$. This lets us write $\delta x = A^{-1}r$, yielding the bound

$$\|\delta x\| = \|A^{-1}r\| \leq \|A^{-1}\| \cdot \|r\|. \tag{2.5}$$

This simple bound is attractive to use in practice, since r is easy to compute, given an approximate solution $\hat{x}$. Furthermore, there is no apparent need to estimate δA and δb. In fact our two approaches are very closely related, as shown by the next theorem.

THEOREM 2.2. *Let $r = A\hat{x} - b$. Then there exists a δA such that $\|\delta A\| = \frac{\|r\|}{\|\hat{x}\|}$ and $(A+\delta A)\hat{x} = b$. No δA of smaller norm and satisfying $(A+\delta A)\hat{x} = b$ exists. Thus, δA is the smallest possible backward error (measured in norm). This is true for any vector norm and its induced norm (or $\|\cdot\|_2$ for vectors and $\|\cdot\|_F$ for matrices).*

Proof. $(A+\delta A)\hat{x} = b$ if and only if $\delta A \cdot \hat{x} = b - A\hat{x} = -r$, so $\|r\| = \|\delta A \cdot \hat{x}\| \leq \|\delta A\| \cdot \|\hat{x}\|$, implying $\|\delta A\| \geq \frac{\|r\|}{\|\hat{x}\|}$. We complete the proof only for the two-norm and its induced matrix norm. Choose $\delta A = \frac{-r\cdot\hat{x}^T}{\|\hat{x}\|_2^2}$. We can easily verify that $\delta A \cdot \hat{x} = -r$ and $\|\delta A\|_2 = \frac{\|r\|_2}{\|\hat{x}\|_2}$. $\square$

Thus, the smallest $\|\delta A\|$ that could yield an $\hat{x}$ satisfying $(A+\delta A)\hat{x} = b$ and $r = A\hat{x} - b$ is given by Theorem 2.2. Applying error bound (2.2) (with $\delta b = 0$) yields

$$\|\delta x\| \leq \|A^{-1}\| \left(\frac{\|r\|}{\|\hat{x}\|} \cdot \|\hat{x}\|\right) = \|A^{-1}\| \cdot \|r\|,$$

the same bound as (2.5).

All our bounds depend on the ability to estimate the condition number $\|A\| \cdot \|A^{-1}\|$. We return to this problem in section 2.4.3. Condition number estimates are computed by LAPACK routines such as `sgesvx`.

2.2.1. Relative Perturbation Theory

In the last section we showed how to bound the norm of the error $\delta x = \hat{x} - x$ in the approximate solution $\hat{x}$ of $Ax = b$. Our bound on $\|\delta x\|$ was proportional to the condition number $\kappa(A) = \|A\| \cdot \|A^{-1}\|$ times the norms $\|\delta A\|$ and $\|\delta b\|$, where $\hat{x}$ satisfies $(A+\delta A)\hat{x} = b + \delta b$.

In many cases this bound is quite satisfactory, but not always. Our goal in this section is to show when it is too pessimistic and to derive an alternative perturbation theory that provides tighter bounds. We will use this perturbation theory later in section 2.5.1 to justify the error bounds computed by the LAPACK subroutines like `sgesvx`.

This section may be skipped on a first reading.

Here is an example where the error bound of the last section is much too pessimistic.

EXAMPLE 2.1. Let $A = \text{diag}(\gamma, 1)$ (a diagonal matrix with entries $a_{11} = \gamma$ and $a_{22} = 1$) and $b = [\gamma, 1]^T$, where $\gamma > 1$. Then $x = A^{-1}b = [1, 1]^T$. Any reasonable direct method will solve $Ax = b$ very accurately (using two divisions b_i/a_{ii}) to get $\hat{x}$, yet the condition number $\kappa(A) = \gamma$ may be arbitrarily large. Therefore our error bound (2.3) may be arbitrarily large.

The reason that the condition number $\kappa(A)$ leads us to overestimate the error is that bound (2.2), from which it comes, assumes that δA is bounded in norm *but is otherwise arbitrary*; this is needed to prove that bound (2.2) is attainable in Question 2.3. In contrast, the δA corresponding to the actual rounding errors is not arbitrary but has a special structure not captured by its norm alone. We can determine the smallest δA corresponding to $\hat{x}$ for our problem as follows: A simple rounding error analysis shows that $\hat{x}_i = (b_i/a_{ii})/(1+\delta_i)$, where $|\delta_i| \leq \varepsilon$. Thus $(a_{ii} + \delta_i a_{ii})\hat{x}_i = b_i$. We may rewrite this

as $(A+\delta A)\hat{x} = b$, where $\delta A = \mathrm{diag}(\delta_1 a_{11}, \delta_2 a_{22})$. Then $\|\delta A\|$ can be as large $\max_i |\varepsilon a_{ii}| = \varepsilon\gamma$. Applying error bound (2.3) with $\delta b = 0$ yields

$$\frac{\|\delta x\|_\infty}{\|\hat{x}\|_\infty} \le \gamma \left(\frac{\varepsilon\gamma}{\gamma}\right) = \varepsilon\gamma.$$

In contrast, the actual error satisfies

$$\begin{aligned} \|\delta x\|_\infty &= \|\hat{x} - x\|_\infty \\ &= \left\| \begin{bmatrix} (b_1/a_{11})/(1+\delta_1) - (b_1/a_{11}) \\ (b_2/a_{22})/(1+\delta_2) - (b_2/a_{22}) \end{bmatrix} \right\|_\infty \\ &= \left\| \begin{bmatrix} -\delta_1/(1+\delta_1) \\ -\delta_2/(1+\delta_2) \end{bmatrix} \right\|_\infty \\ &\le \frac{\varepsilon}{1-\varepsilon} \end{aligned}$$

or

$$\frac{\|\delta x\|_\infty}{\|\hat{x}\|_\infty} \le \varepsilon/(1-\varepsilon)^2,$$

which is about γ times smaller. $\diamond$

For this example, we can describe the structure of the actual δA as follows: $|\delta a_{ij}| \le \epsilon |a_{ij}|$, where ϵ is a tiny number. We write this more succinctly as

$$|\delta A| \le \epsilon |A| \tag{2.6}$$

(see section 1.1 for notation). We also say that δA is a *small componentwise relative perturbation in A*. Since δA can often be made to satisfy bound (2.6) in practice, along with $|\delta b| \le \epsilon |b|$ (see section 2.5.1), we will derive perturbation theory using these bounds on δA and δb.

We begin with equation (2.1):

$$\delta x = A^{-1}(-\delta A \hat{x} + \delta b).$$

Now take absolute values, and repeatedly use the triangle inequality to get

$$\begin{aligned} |\delta x| &= |A^{-1}(-\delta A\hat{x} + \delta b)| \\ &\le |A^{-1}|(|\delta A| \cdot |\hat{x}| + |\delta b|) \\ &\le |A^{-1}|(\epsilon |A| \cdot |\hat{x}| + \epsilon |b|) \\ &= \epsilon(|A^{-1}|(|A| \cdot |\hat{x}| + |b|)). \end{aligned}$$

Now using any vector norm (like the infinity-, one-, or Frobenius norms), where $\| \, |z| \, \| = \|z\|$, we get the bound

$$\|\delta x\| \le \epsilon \| |A^{-1}|(|A| \cdot |\hat{x}| + |b|)\|. \tag{2.7}$$

Assuming for the moment that $\delta b = 0$, we can weaken this bound to

$$\|\delta x\| \le \epsilon \| |A^{-1}| \cdot |A| \| \cdot \|\hat{x}\|$$

or

$$\frac{\|\delta x\|}{\|x\|} \le \epsilon \| \, |A^{-1}| \cdot |A| \, \|. \tag{2.8}$$

This leads us to define $\kappa_{CR}(A) \equiv \| \, |A^{-1}| \cdot |A| \, \|$ as the *componentwise relative condition number of A*, or just *relative condition number* for short. It is sometimes also called the Bauer condition number [26] or Skeel condition number [225, 226, 227]. For a proof that bounds (2.7) and (2.8) are attainable, see Question 2.4.

Recall that Theorem 2.1 related the condition number $\kappa(A)$ to the distance from A to the nearest singular matrix. For a similar interpretation of $\kappa_{CR}(A)$, see [72, 208].

EXAMPLE 2.2. Consider our earlier example with $A = \mathrm{diag}(\gamma, 1)$ and $b = [\gamma, 1]^T$. It is easy to confirm that $\kappa_{CR}(A) = 1$, since $|A^{-1}| \cdot |A| = I$. Indeed, $\kappa_{CR}(A) = 1$ for any diagonal matrix A, capturing our intuition that a diagonal system of equations should be solvable quite accurately. $\diamond$

More generally, suppose that D is any nonsingular diagonal matrix and B is an arbitrary nonsingular matrix. Then

$$\begin{aligned} \kappa_{CR}(DB) &= \| \, |(DB)^{-1}| \cdot |(DB)| \, \| \\ &= \| \, |B^{-1}D^{-1}| \cdot |DB| \, \| \\ &= \| \, |B^{-1}| \cdot |B| \, \| \\ &= \kappa_{CR}(B). \end{aligned}$$

This means that if DB is *badly scaled*, i.e., B is well-conditioned but DB is badly conditioned (because D has widely varying diagonal entries), then we should hope to get an accurate solution of $(DB)x = b$ despite DB's ill-conditioning. This is discussed further in sections 2.4.4, 2.5.1, and 2.5.2.

Finally, as in the last section we provide an error bound using only the residual $r = A\hat{x} - b$:

$$\|\delta x\| = \|A^{-1}r\| \le \| \, |A^{-1}| \cdot |r| \, \|, \tag{2.9}$$

where we have used the triangle inequality. In section 2.4.4 we will see that this bound can sometimes be much smaller than the similar bound (2.5), in particular when A is badly scaled. There is also an analogue to Theorem 2.2 [193].

THEOREM 2.3. *The smallest $\epsilon > 0$ such that there exist $|\delta A| \le \epsilon |A|$ and $|\delta b| \le \epsilon |b|$ satisfying $(A+\delta A)\hat{x} = b + \delta b$ is called the* componentwise relative backward error. *It may be expressed in terms of the residual $r = A\hat{x} - b$ as follows:*

$$\epsilon = \max_i \frac{|r_i|}{(|A| \cdot |\hat{x}| + |b|)_i}.$$

For a proof, see Question 2.5.

LAPACK routines like sgesvx compute the componentwise backward relative error ϵ (the LAPACK variable name for ϵ is BERR).

2.3. Gaussian Elimination

The basic algorithm for solving $Ax = b$ is *Gaussian elimination*. To state it, we first need to define a *permutation matrix*.

DEFINITION 2.1. *A* permutation matrix P *is an identity matrix with permuted rows.*

The most important properties of a permutation matrix are given by the following lemma.

LEMMA 2.2. *Let P, P_1, and P_2 be n-by-n permutation matrices and X be an n-by-n matrix. Then*

1. *PX is the same as X with its rows permuted. XP is the same as X with its columns permuted.*
2. $P^{-1} = P^T$.
3. $\det(P) = \pm 1$.
4. *$P_1 \cdot P_2$ is also a permutation matrix.*

For a proof, see Question 2.6.

Now we can state our overall algorithm for solving $Ax = b$.

ALGORITHM 2.1. *Solving $Ax = b$ using Gaussian elimination:*

1. *Factorize A into $A = PLU$, where*
   P = *permutation matrix,*
   L = *unit lower triangular matrix (i.e., with ones on the diagonal),*
   U = *nonsingular upper triangular matrix.*
2. *Solve $PLUx = b$ for LUx by permuting the entries of b: $LUx = P^{-1}b = P^Tb$.*
3. *Solve $LUx = P^{-1}b$ for Ux by forward substitution: $Ux = L^{-1}(P^{-1}b)$.*
4. *Solve $Ux = L^{-1}(P^{-1}b)$ for x by back substitution: $x = U^{-1}(L^{-1}P^{-1}b)$.*

We will derive the algorithm for factorizing $A = PLU$ in several ways. We begin by showing why the permutation matrix P is necessary.

DEFINITION 2.2. *The* leading j-by-j principal submatrix *of A is $A(1:j, 1:j)$.*

THEOREM 2.4. *The following two statements are equivalent:*

1. *There exists a unique unit lower triangular L and nonsingular upper triangular U such that $A = LU$.*
2. *All leading principal submatrices of A are nonsingular.*

Proof. We first show (1) implies (2). $A = LU$ may also be written

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} = \begin{bmatrix} L_{11} & 0 \\ L_{21} & L_{22} \end{bmatrix} \begin{bmatrix} U_{11} & U_{12} \\ 0 & U_{22} \end{bmatrix} = \begin{bmatrix} L_{11}U_{11} & L_{11}U_{12} \\ L_{21}U_{11} & L_{21}U_{12} + L_{22}U_{22} \end{bmatrix},$$

where A_{11} is a j-by-j leading principal submatrix, as are L_{11} and U_{11}. Therefore $\det A_{11} = \det(L_{11}U_{11}) = \det L_{11} \det U_{11} = 1 \cdot \prod_{k=1}^{j}(U_{11})_{kk} \neq 0$, since L is unit triangular and U is triangular.

We prove that (2) implies (1) by induction on n. It is easy for 1-by-1 matrices: $a = 1 \cdot a$. To prove it for n-by-n matrices $\tilde{A}$, we need to find unique $(n-1)$-by-$(n-1)$ triangular matrices L and U, unique $(n-1)$-by-1 vectors l and u, and a unique nonzero scalar η such that

$$\tilde{A} = \begin{bmatrix} A & b \\ c^T & \delta \end{bmatrix} = \begin{bmatrix} L & 0 \\ l^T & 1 \end{bmatrix} \begin{bmatrix} U & u \\ 0 & \eta \end{bmatrix} = \begin{bmatrix} LU & Lu \\ l^TU & l^Tu + \eta \end{bmatrix}.$$

By induction, unique L and U exist such that $A = LU$. Now let $u = L^{-1}b$, $l^T = c^TU^{-1}$, and $\eta = \delta - l^Tu$, all of which are unique. The diagonal entries of U are nonzero by induction, and $\eta \neq 0$ since $0 \neq \det(\tilde{A}) = \det(U) \cdot \eta$. □

Thus LU factorization without pivoting can fail on (well-conditioned) nonsingular matrices such as the permutation matrix

$$P = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix};$$

the 1-by-1 and 2-by-2 leading principal minors of P are singular. So we need to introduce permutations into Gaussian elimination.

THEOREM 2.5. *If A is nonsingular, then there exist permutations P_1 and P_2, a unit lower triangular matrix L, and a nonsingular upper triangular matrix U such that $P_1AP_2 = LU$. Only one of P_1 and P_2 is necessary.*

Note: P_1A reorders the rows of A, AP_2 reorders the columns, and P_1AP_2 reorders both.

Proof. As with many matrix factorizations, it suffices to understand block 2-by-2 matrices. More formally, we use induction on the dimension n. It is easy for 1-by-1 matrices: $P_1 = P_2 = L = 1$ and $U = A$. Assume that it is true for dimension $n-1$. If A is nonsingular, then it has a nonzero entry; choose permutations P_1' and P_2' so that the $(1,1)$ entry of $P_1'AP_2'$ is nonzero. (We need only one of P_1' and P_2' since nonsingularity implies that each row and each column of A has a nonzero entry.)

Now we write the desired factorization and solve for the unknown components:

$$\begin{aligned} P_1'AP_2' &= \begin{bmatrix} a_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ L_{21} & I \end{bmatrix} \cdot \begin{bmatrix} u_{11} & U_{12} \\ 0 & \tilde{A}_{22} \end{bmatrix} \\ &= \begin{bmatrix} u_{11} & U_{12} \\ L_{21}u_{11} & L_{21}U_{12} + \tilde{A}_{22} \end{bmatrix}, \end{aligned} \tag{2.10}$$

where A_{22} and $\tilde{A}_{22}$ are $(n-1)$-by-$(n-1)$ and L_{21} and U_{12}^T are $(n-1)$-by-1.

Solving for the components of this 2-by-2 block factorization we get $u_{11} = a_{11} \neq 0$, $U_{12} = A_{12}$, and $L_{21}u_{11} = A_{21}$. Since $u_{11} = a_{11} \neq 0$, we can solve for $L_{21} = \frac{A_{21}}{a_{11}}$. Finally, $L_{21}U_{12} + \tilde{A}_{22} = A_{22}$ implies $\tilde{A}_{22} = A_{22} - L_{21}U_{12}$.

We want to apply induction to $\tilde{A}_{22}$, but to do so we need to check that $\det \tilde{A}_{22} \neq 0$: Since $\det P_1'AP_2' = \pm \det A \neq 0$ and also

$$\det P_1'AP_2' = \det \begin{bmatrix} 1 & 0 \\ L_{21} & I \end{bmatrix} \cdot \det \begin{bmatrix} u_{11} & U_{12} \\ 0 & \tilde{A}_{22} \end{bmatrix} = 1 \cdot (u_{11} \cdot \det \tilde{A}_{22}),$$

then $\det \tilde{A}_{22}$ must be nonzero.

Therefore, by induction there exist permutations $\tilde{P}_1$ and $\tilde{P}_2$ so that $\tilde{P}_1\tilde{A}_{22}\tilde{P}_2 = \tilde{L}\tilde{U}$, with $\tilde{L}$ unit lower triangular and $\tilde{U}$ upper triangular and nonsingular. Substituting this in the above 2-by-2 block factorization yields

$$\begin{aligned} P_1'AP_2' &= \begin{bmatrix} 1 & 0 \\ L_{21} & I \end{bmatrix} \begin{bmatrix} u_{11} & U_{12} \\ 0 & \tilde{P}_1^T\tilde{L}\tilde{U}\tilde{P}_2^T \end{bmatrix} \\ &= \begin{bmatrix} 1 & 0 \\ L_{21} & I \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & \tilde{P}_1^T\tilde{L} \end{bmatrix} \begin{bmatrix} u_{11} & U_{12} \\ 0 & \tilde{U}\tilde{P}_2^T \end{bmatrix} \\ &= \begin{bmatrix} 1 & 0 \\ L_{21} & \tilde{P}_1^T\tilde{L} \end{bmatrix} \begin{bmatrix} u_{11} & U_{12}\tilde{P}_2 \\ 0 & \tilde{U} \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & \tilde{P}_2^T \end{bmatrix} \\ &= \begin{bmatrix} 1 & 0 \\ 0 & \tilde{P}_1^T \end{bmatrix} \begin{bmatrix} 1 & 0 \\ \tilde{P}_1L_{21} & \tilde{L} \end{bmatrix} \begin{bmatrix} u_{11} & U_{12}\tilde{P}_2 \\ 0 & \tilde{U} \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & \tilde{P}_2^T \end{bmatrix}, \end{aligned}$$

so we get the desired factorization of A:

$$\begin{aligned} P_1AP_2 &= \left(\begin{bmatrix} 1 & 0 \\ 0 & \tilde{P}_1 \end{bmatrix} P_1' \right) A \left(P_2' \begin{bmatrix} 1 & 0 \\ 0 & \tilde{P}_2 \end{bmatrix} \right) \\ &= \begin{bmatrix} 1 & 0 \\ \tilde{P}_1L_{21} & \tilde{L} \end{bmatrix} \begin{bmatrix} u_{11} & U_{12}\tilde{P}_2 \\ 0 & \tilde{U} \end{bmatrix}. \quad \square \end{aligned}$$

The next two corollaries state simple ways to choose P_1 and P_2 to guarantee that Gaussian elimination will succeed on a nonsingular matrix.

COROLLARY 2.1. *We can choose $P_2' = I$ and P_1' so that a_{11} is the largest entry in absolute value in its column, which implies $L_{21} = \frac{A_{21}}{a_{11}}$ has entries bounded by 1 in absolute value. More generally, at step i of Gaussian elimination, where we are computing the ith column of L, we reorder rows i through n so that the largest entry in the column is on the diagonal. This is called "Gaussian elimination with partial pivoting," or GEPP for short. GEPP guarantees that all entries of L are bounded by one in absolute value.*

GEPP is the most common way to implement Gaussian elimination in practice. We discuss its numerical stability in the next section. Another more expensive way to choose P_1 and P_2 is given by the next corollary. It is almost never used in practice, although there are rare examples where GEPP fails but the next method succeeds in computing an accurate answer (see Question 2.14). We discuss briefly it in the next section as well.

COROLLARY 2.2. *We can choose P_1' and P_2' so that a_{11} is the largest entry in absolute value in the whole matrix. More generally, at step i of Gaussian elimination, where we are computing the ith column of L, we reorder rows and columns i through n so that the largest entry in this submatrix is on the diagonal. This is called "Gaussian elimination with complete pivoting," or GECP for short.*

The following algorithm embodies Theorem 2.5, performing permutations, computing the first column of L and the first row of U, and updating A_{22} to get $\tilde{A}_{22} = A_{22} - L_{21}U_{12}$. We write the algorithm first in conventional programming language notation and then using Matlab notation.

ALGORITHM 2.2. *LU factorization with pivoting:*

```
for i = 1 to n-1
    apply permutations so a_ii ≠ 0 (permute L and U too)
        /* for example, for GEPP, swap rows j and i of A and of L
             where |a_ji| is the largest entry in |A(i:n,i)|;
             for GECP, swap rows j and i of A and of L,
             and columns k and i of A and of U,
             where |a_jk| is the largest entry in |A(i:n,i:n)| */
    /* compute column i of L (L_21 in (2.10)) */
    for j = i+1 to n
        l_ji = a_ji/a_ii
    end for
    /* compute row i of U (U_12 in (2.10)) */
    for j = i to n
```

```
        u_ij = a_ij
    end for
    /* update A_22 (to get Ã_22 = A_22 − L_21 U_12 in (2.10)) */
    for j = i + 1 to n
        for k = i + 1 to n
            a_jk = a_jk − l_ji * u_ik
        end for
    end for
end for
```

Note that once column i of A is used to compute column i of L, it is never used again. Similarly, row i of A is never used again after computing row i of U. This lets us overwrite L and U on top of A as they are computed, so we need no extra space to store them; L occupies the (strict) lower triangle of A (the ones on the diagonal of L are not stored explicitly), and U occupies the upper triangle of A. This simplifies the algorithm to the following algorithm.

ALGORITHM 2.3. *LU factorization with pivoting, overwriting L and U on A:*

```
for i = 1 to n − 1
    apply permutations (see Algorithm 2.2 for details)
    for j = i + 1 to n
        a_ji = a_ji/a_ii
    end for
    for j = i + 1 to n
        for k = i + 1 to n
            a_jk = a_jk − a_ji * a_ik
        end for
    end for
end for
```

Using Matlab notation this further reduces to the following algorithm.

ALGORITHM 2.4. *LU factorization with pivoting, overwriting L and U on A:*

```
for i = 1 to n − 1
    apply permutations (see Algorithm 2.2 for details)
    A(i + 1 : n, i) = A(i + 1 : n, i)/A(i, i)
    A(i + 1 : n, i + 1 : n) =
        A(i + 1 : n, i + 1 : n) − A(i + 1 : n, i) * A(i, i + 1 : n)
end for
```

In the last line of the algorithm, $A(i+1:n,i)*A(i,i+1:n)$ is the product of an $(n-i)$-by-1 matrix (L_{21}) by a 1-by-$(n-i)$ matrix (U_{12}), which yields an $(n-i)$-by-$(n-i)$ matrix.

We now rederive this algorithm from scratch starting from perhaps the most familiar description of Gaussian elimination: "Take each row and subtract multiples of it from later rows to zero out the entries below the diagonal." Translating this directly into an algorithm yields

```
for i = 1 to n − 1                  /* for each row i */
    for j = i + 1 to n              /* subtract a multiple of
                                       row i from row j ... */
        for k = i to n              /* ... in columns i through n ... */
            a_jk = a_jk − (a_ji/a_ii) a_ik   /* ... to zero out column i
                                                below the diagonal */
        end for
    end for
end for
```

We will now make some improvements to this algorithm, modifying it until it becomes identical to Algorithm 2.3 (except for pivoting, which we omit). First, we recognize that we need not compute the zero entries below the diagonal, because we know they are zero. This shortens the k loop to yield

```
for i = 1 to n − 1
    for j = i + 1 to n
        for k = i + 1 to n
            a_jk = a_jk − (a_ji/a_ii) a_ik
        end for
    end for
end for
```

The next performance improvement is to compute $\frac{a_{ji}}{a_{ii}}$ outside the inner loop, since it is constant within the inner loop.

```
for i = 1 to n − 1
    for j = i + 1 to n
        l_ji = a_ji/a_ii
    end for
    for j = i + 1 to n
        for k = i + 1 to n
            a_jk = a_jk − l_ji a_ik
        end for
    end for
end for
```

Finally, we store the multipliers l_{ji} in the subdiagonal entries a_{ji} that we originally zeroed out; they are not needed for anything else. This yields Algorithm 2.3 (except for pivoting).

The operation count of LU is done by replacing loops by summations over the same range, and inner loops by their operation counts:

$$\sum_{i=1}^{n-1}\left(\sum_{j=i+1}^{n} 1 + \sum_{j=i+1}^{n}\sum_{k=i+1}^{n} 2\right)$$
$$= \sum_{i=1}^{n-1}((n-i)+2(n-i)^2) = \frac{2}{3}n^3 + O(n^2).$$

The forward and back substitutions with L and U to complete the solution of $Ax = b$ cost $O(n^2)$, so overall solving $Ax = b$ with Gaussian elimination costs $\frac{2}{3}n^3 + O(n^2)$ operations. Here we have used the fact that $\sum_{i=1}^{m} i^k = m^{k+1}/(k+1) + O(m^k)$. This formula is enough to get the high-order term in the operation count.

There is more to implementing Gaussian elimination than writing the nested loops of Algorithm 2.2. Indeed, depending on the computer, programming language, and matrix size, merely interchanging the last two loops on j and k can change the execution time by orders of magnitude. We discuss this at length in section 2.6.

2.4. Error Analysis

Recall our two-step paradigm for obtaining error bounds for the solution of $Ax = b$:

1. Analyze roundoff errors to show that the result of solving $Ax = b$ is the exact solution $\hat{x}$ of the perturbed linear system $(A+\delta A)\hat{x} = b+\delta b$, where δA and δb are small. This is an example of *backward error analysis*, and δA and δb are called the *backward errors*.

2. Apply the perturbation theory of section 2.2 to bound the error, for example by using bound (2.3) or (2.5).

We have two goals in this section. The first is to show how to implement Gaussian elimination in order to keep the backward errors δA and δb small. In particular, we would like to keep $\frac{\|\delta A\|}{\|A\|}$ and $\frac{\|\delta b\|}{\|b\|}$ as small as $O(\varepsilon)$. This is as small as we can expect to make them, since merely rounding the largest entries of A (or b) to fit into the floating point format can make $\frac{\|\delta A\|}{\|A\|} \geq \varepsilon$ (or $\frac{\|\delta b\|}{\|b\|} \geq \varepsilon$). It turns out that unless we are careful about pivoting, δA and δb need not be small. We discuss this in the next section.

The second goal is to derive practical error bounds which are simultaneously cheap to compute and "tight," i.e., close to the true errors. It turns out that the best bounds for $\|\delta A\|$ that we can formally prove are generally much larger than the errors encountered in practice. Therefore, our practical error bounds

(in section 2.4.4) will rely on the computed residual $r = A\hat{x} - b$ and bound (2.5), instead of bound (2.3). We also need to be able to estimate $\kappa(A)$ inexpensively; this is discussed in section 2.4.3.

Unfortunately, we do not have error bounds that *always* satisfy our twin goals of cheapness and tightness, i.e., that simultaneously

1. cost a negligible amount compared to solving $Ax = b$ in the first place (for example, that cost $O(n^2)$ flops versus Gaussian elimination's $O(n^3)$ flops),

2. provide an error bound that is always at least as large as the true error and never more than a constant factor larger (100 times larger, say).

The practical bounds in section 2.4.4 will cost $O(n^2)$ but will on very rare occasions provide error bounds that are much too small or much too large. The probability of getting a bad error bound is so small that these bounds are widely used in practice. The only truly guaranteed bounds use either interval arithmetic, very high precision arithmetic, or both, and are several times more expensive than just solving $Ax = b$ (see section 1.5).

It has in fact been conjectured that no bound satisfying our twin goals of cheapness and tightness exist, but this remains an open problem.

2.4.1. The Need for Pivoting

Let us apply LU factorization without pivoting to $A = \begin{bmatrix} .0001 & 1 \\ 1 & 1 \end{bmatrix}$ in three-decimal-digit floating point arithmetic and see why we get the wrong answer. Note that $\kappa(A) = \|A\|_\infty \cdot \|A^{-1}\|_\infty \approx 4$, so A is well conditioned and thus we should expect to be able to solve $Ax = b$ accurately.

$$L = \begin{bmatrix} 1 & 0 \\ \mathrm{fl}(1/10^{-4}) & 1 \end{bmatrix}, \ \mathrm{fl}(1/10^{-4}) \text{ rounds to } 10^4,$$
$$U = \begin{bmatrix} 10^{-4} & 1 \\ & \mathrm{fl}(1-10^4\cdot 1) \end{bmatrix}, \ \mathrm{fl}(1-10^4\cdot 1) \text{ rounds to } -10^4,$$
$$\text{so } LU = \begin{bmatrix} 1 & 0 \\ 10^4 & 1 \end{bmatrix}\begin{bmatrix} 10^{-4} & 1 \\ & -10^4 \end{bmatrix} = \begin{bmatrix} 10^{-4} & 1 \\ 1 & 0 \end{bmatrix}$$
$$\text{but } A = \begin{bmatrix} 10^{-4} & 1 \\ 1 & 1 \end{bmatrix}.$$

Note that the original a_{22} has been entirely "lost" from the computation by subtracting 10^4 from it. We would have gotten the same LU factors whether a_{22} had been 1, 0, -2, or any number such that $\mathrm{fl}(a_{22} - 10^4) = -10^4$. Since the algorithm proceeds to work only with L and U, it will get the same answer for all these different a_{22}, which correspond to completely different A and so completely different $x = A^{-1}b$; there is no way to guarantee an accurate answer. This is called *numerical instability*, since L and U are *not* the exact

factors of a matrix close to A. (Another way to say this is that $\|A - LU\|$ is about as large as $\|A\|$, rather than $\varepsilon\|A\|$.)

Let us see what happens when we go on to solve $Ax = [1,2]^T$ for x using this LU factorization. The correct answer is $x \approx [1,1]^T$. Instead we get the following. Solving $Ly = [1,2]^T$ yields $y_1 = \mathrm{fl}(1/1) = 1$ and $y_2 = \mathrm{fl}(2 - 10^4 \cdot 1) = -10^4$; note that the value 2 has been "lost" by subtracting 10^4 from it. Solving $U\hat{x} = y$ yields $\hat{x}_2 = \mathrm{fl}((-10^4)/(-10^4)) = 1$ and $\hat{x}_1 = \mathrm{fl}((1-1)/10^{-4}) = 0$, a completely erroneous solution.

Another warning of the loss of accuracy comes from comparing the condition number of A to the condition numbers of L and U. Recall that we transform the problem of solving $Ax = b$ into solving two other systems with L and U, so we do not want the condition numbers of L or U to be much larger than that of A. But here, the condition number of A is about 4, whereas the condition numbers of L and U are about 10^8.

In the next section we will show that doing GEPP nearly always eliminates the instability just illustrated. In the above example, GEPP would have reversed the order of the two equations before proceeding. The reader is invited to confirm that in this case we would get

$$L = \begin{bmatrix} 1 & 0 \\ \mathrm{fl}(.0001/1) & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ .0001 & 1 \end{bmatrix}$$

and

$$U = \begin{bmatrix} 1 & 1 \\ 0 & \mathrm{fl}(1 - .0001 \cdot 1) \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}$$

so that LU approximates A quite accurately. Both L and U are quite well-conditioned, as is A. The computed solution vector is also quite accurate.

2.4.2. Formal Error Analysis of Gaussian Elimination

Here is the intuition behind our error analysis of LU decomposition. If intermediate quantities arising in the product $L \cdot U$ are very large compared to $\|A\|$, the information in entries of A will get "lost" when these large values are subtracted from them. This is what happened to a_{22} in the example in section 2.4.1. If the intermediate quantities in the product $L \cdot U$ were instead comparable to those of A, we would expect a tiny backward error $A - LU$ in the factorization. Therefore, we want to bound the largest intermediate quantities in the product $L \cdot U$. We will do this by bounding the entries of the matrix $|L| \cdot |U|$ (see section 1.1 for notation).

Our analysis is analogous to the one we used for polynomial evaluation in section 1.6. There we considered $p = \sum_i a_i x^i$ and showed that if $|p|$ were comparable to the sum of absolute values $\sum_i |a_i x^i|$, then p would be computed accurately.

After presenting a general analysis of Gaussian elimination, we will use it to show that GEPP (or, more expensively, GECP) will keep the entries of $|L| \cdot |U|$ comparable to $\|A\|$ in almost all practical circumstances.

Unfortunately, the best bounds on $\|\delta A\|$ that we can prove in general are still much larger than the errors encountered in practice. Therefore, the error bounds that we use in practice will be based on the computed residual r and bound (2.5) (or bound (2.9)) instead of the rigorous but pessimistic bound in this section.

Now suppose that matrix A has already been pivoted, so the notation is simpler. We simplify Algorithm 2.2 to two equations, one for a_{jk} with $j \le k$ and one for $j > k$. Let us first trace what Algorithm 2.2 does to a_{jk} when $j \le k$: this element is repeatedly updated by subtracting $l_{ji}u_{ik}$ for $i = 1$ to $j-1$ and is finally assigned to u_{jk} so that

$$u_{jk} = a_{jk} - \sum_{i=1}^{j-1} l_{ji}u_{ik}.$$

When $j > k$, a_{jk} again has $l_{ji}u_{ik}$ subtracted for $i = 1$ to $k-1$, and then the resulting sum is divided by u_{kk} and assigned to l_{jk}:

$$l_{jk} = \frac{a_{jk} - \sum_{i=1}^{k-1} l_{ji}u_{ik}}{u_{kk}}.$$

To do the roundoff error analysis of these two formulas, we use the result from Question 1.10 that a dot product computed in floating point arithmetic satisfies

$$\mathrm{fl}\left(\sum_{i=1}^{d} x_i y_i\right) = \sum_{i=1}^{d} x_i y_i (1+\delta_i) \quad \text{with } |\delta_i| \le d\varepsilon.$$

We apply this to the formula for u_{jk}, yielding[9]

$$u_{jk} = \left(a_{jk} - \sum_{i=1}^{j-1} l_{ji}u_{ik}(1+\delta_i)\right)(1+\delta')$$

with $|\delta_i| \le (j-1)\varepsilon$ and $|\delta'| \le \varepsilon$. Solving for a_{jk} we get

$$\begin{aligned} a_{jk} &= \tfrac{1}{1+\delta'} u_{jk} \cdot l_{jj} + \sum_{i=1}^{j-1} l_{ji}u_{ik}(1+\delta_i) \quad \text{since } l_{jj} = 1 \\ &= \sum_{i=1}^{j} l_{ji}u_{ik} + \sum_{i=1}^{j} l_{ji}u_{ik}\delta_i \\ &\quad \text{with } |\delta_i| \le (j-1)\varepsilon \text{ and } 1+\delta_j \equiv \tfrac{1}{1+\delta'} \\ &\equiv \sum_{i=1}^{j} l_{ji}u_{ik} + E_{jk}, \end{aligned}$$

[9] Strictly speaking, the next formula assumes that we compute the sum first and then subtract from a_{jk}. But the final bound does not depend on the order of summation.

where we can bound E_{jk} by

$$|E_{jk}| = \left| \sum_{i=1}^{j} l_{ji} \cdot u_{ik} \cdot \delta_i \right| \le \sum_{i=1}^{j} |l_{ji}| \cdot |u_{ik}| \cdot n\varepsilon = n\varepsilon (|L| \cdot |U|)_{jk}.$$

- Doing the same analysis for the formula for l_{jk} yields

$$l_{jk} = (1+\delta'') \left(\frac{(1+\delta')(a_{jk} - \sum_{i=1}^{k-1} l_{ji} u_{ik}(1+\delta_i))}{u_{kk}} \right)$$

with $|\delta_i| \le (k-1)\varepsilon$, $|\delta'| \le \varepsilon$, and $|\delta''| \le \varepsilon$. We solve for a_{jk} to get

$$\begin{aligned} a_{jk} &= \frac{1}{(1+\delta')(1+\delta'')} u_{kk} l_{jk} + \sum_{i=1}^{k-1} l_{ji} u_{ik}(1+\delta_i) \\ &= \sum_{i=1}^{k} l_{ji} u_{ik} + \sum_{i=1}^{k} l_{ji} u_{ik} \delta_i \quad \text{with } 1+\delta_k \equiv \frac{1}{(1+\delta')(1+\delta'')} \\ &\equiv \sum_{i=1}^{k} l_{ji} u_{ik} + E_{jk} \end{aligned}$$

with $|\delta_i| \le n\varepsilon$, and so $|E_{jk}| \le n\varepsilon(|L| \cdot |U|)_{jk}$ as before.

Altogether, we can summarize this error analysis with the simple formula $A = LU + E$ where $|E| \le n\varepsilon |L| \cdot |U|$. Taking norms we get $\|E\| \le n\varepsilon \| \, |L| \, \| \cdot \| \, |U| \, \|$. If the norm does not depend on the signs of the matrix entries (true for the Frobenius, infinity-, and one-norms but not the two-norm), we can simplify this to $\|E\| \le n\varepsilon \|L\| \cdot \|U\|$.

Now we consider the rest of the problem: solving $LUx = b$ via $Ly = b$ and $Ux = y$. The result of Question 1.11 shows that solving $Ly = b$ by forward substitution yields a computed solution $\hat{y}$ satisfying $(L+\delta L)\hat{y} = b$ with $|\delta L| \le n\varepsilon |L|$. Similarly when solving $Ux = \hat{y}$ we get $\hat{x}$ satisfying $(U+\delta U)\hat{x} = \hat{y}$ with $|\delta U| \le n\varepsilon |U|$.

Combining these yields

$$\begin{aligned} b &= (L+\delta L)\hat{y} \\ &= (L+\delta L)(U+\delta U)\hat{x} \\ &= (LU + L\delta U + \delta L U + \delta L \delta U)\hat{x} \\ &= (A - E + L\delta U + \delta L U + \delta L \delta U)\hat{x} \\ &\equiv (A+\delta A)\hat{x}, \quad \text{where } \delta A = -E + L\delta U + \delta L U + \delta L \delta U. \end{aligned}$$

Now we combine our bounds on E, δL, and δU and use the triangle inequality to bound δA:

$$\begin{aligned} |\delta A| &= |-E + L\delta U + \delta L U + \delta L \delta U| \\ &\le |E| + |L\delta U| + |\delta L U| + |\delta L \delta U| \\ &\le |E| + |L| \cdot |\delta U| + |\delta L| \cdot |U| + |\delta L| \cdot |\delta U| \\ &\le n\varepsilon |L| \cdot |U| + n\varepsilon |L| \cdot |U| + n\varepsilon |L| \cdot |U| + n^2\varepsilon^2 |L| \cdot |U| \\ &\approx 3n\varepsilon |L| \cdot |U|. \end{aligned}$$

Taking norms and assuming $\| \, |X| \, \| = \|X\|$ (true as before for the Frobenius, infinity-, and one-norms but not the two-norm) we get $\|\delta A\| \le 3n\varepsilon \|L\| \cdot \|U\|$.

Thus, to see when Gaussian elimination is backward stable, we must ask when $3n\varepsilon \|L\| \cdot \|U\| = O(\varepsilon)\|A\|$; then the $\frac{\|\delta A\|}{\|A\|}$ in the perturbation theory bounds will be $O(\varepsilon)$ as we desire (note that $\delta b = 0$).

The main empirical observation, justified by decades of experience, is that GEPP *almost* always keeps $\|L\| \cdot \|U\| \approx \|A\|$. GEPP guarantees that each entry of L is bounded by 1 in absolute value, so we need consider only $\|U\|$. We define the *pivot growth factor for GEPP*[10] as $g_{\text{PP}} = \|U\|_{\max} / \|A\|_{\max}$, where $\|A\|_{\max} = \max_{ij} |a_{ij}|$, so stability is equivalent to g_{PP} being small or growing slowly as a function of n. In practice, g_{PP} is almost always n or less. The average behavior seems to be $n^{2/3}$ or perhaps even just $n^{1/2}$ [242]. (See Figure 2.1.) This makes GEPP the algorithm of choice for many problems. Unfortunately, there are rare examples in which g_{PP} can be as large as 2^{n-1}.

PROPOSITION 2.1. *GEPP guarantees that $g_{\text{PP}} \le 2^{n-1}$. This bound is attainable.*

Proof. The first step of GEPP updates $\tilde{a}_{jk} = a_{jk} - l_{ji} \cdot u_{ik}$, where $|l_{ji}| \le 1$ and $|u_{ik}| = |a_{ik}| \le \max_{rs} |a_{rs}|$, so $|\tilde{a}_{jk}| \le 2 \cdot \max_{rs} |a_{rs}|$. So each of the $n-1$ major steps of GEPP can double the size of the remaining matrix entries, and we get 2^{n-1} as the overall bound. See the example in Question 2.14 to see that this is attainable. □

Putting all these bounds together, we get

$$\|\delta A\|_\infty \le 3 g_{\text{PP}} n^3 \varepsilon \|A\|_\infty, \tag{2.11}$$

since $\|L\|_\infty \le n$ and $\|U\|_\infty \le n g_{\text{PP}} \|A\|_\infty$. The factor $3 g_{\text{PP}} n^3$ in the bound causes it to almost always greatly overestimate the true $\|\delta A\|$, even if $g_{\text{PP}} = 1$. For example, if $\varepsilon = 10^{-7}$ and $n = 150$, a very modest-sized matrix, then $3n^3\varepsilon > 1$, meaning that all precision is potentially lost. Example 2.3 graphs $3 g_{\text{PP}} n^3 \varepsilon$ along with the true backward error to show how it can be pessimistic; $\|\delta A\|$ is usually $O(\varepsilon)\|A\|$, so we can say that GEPP is *backward stable in practice*, even though we can construct examples where it fails. Section 2.4.4 presents practical error bounds for the computed solution of $Ax = b$ that are much smaller than what we get from using $\|\delta A\|_\infty \le 3 g_{\text{PP}} n^3 \varepsilon \|A\|_\infty$.

[10] This definition is slightly different from the usual one in the literature but essentially equivalent [121, p. 115].

It can be shown that GECP is even more stable than GEPP, with its pivot growth g_{CP} satisfying the worst-case bound [262, p. 213]

$$g_{\text{CP}} = \frac{\max_{ij} |u_{ij}|}{\max_{ij} |a_{ij}|} \leq \sqrt{n \cdot 2 \cdot 3^{1/2} \cdot 4^{1/3} \cdots n^{1/(n-1)}} \approx n^{1/2+\log_e n/4}.$$

This upper bound is also much too large in practice. The average behavior of g_{CP} is $n^{1/2}$. It was an old open conjecture that $g_{\text{CP}} \leq n$, but this was recently disproved [99, 122]. It remains an open problem to find a good upper bound for g_{CP} (which is still widely suspected to be $O(n)$.)

The extra $O(n^3)$ comparisons that GECP uses to find the pivots ($O(n^2)$ comparisons per step, versus $O(n)$ for GEPP) makes GECP significantly slower than GEPP, especially on high-performance machines that perform floating point operations about as fast as comparisons. Therefore, using GECP is seldom warranted (but see sections 2.4.4, 2.5.1, and 5.4.3).

EXAMPLE 2.3. Figures 2.1 and 2.2 illustrate these backward error bounds. For both figures, five random matrices A of each dimension were generated, with independent normally distributed entries, of mean 0 and standard deviation 1. (Testing such random matrices can sometimes be misleading about the behavior on some real problems, but it is still informative.) For each matrix, a similarly random vector b was generated. Both GEPP and GECP were used to solve $Ax = b$. Figure 2.1 plots the pivot growth factors g_{PP} and g_{CP}. In both cases they grow slowly with dimension, as expected. Figure 2.2 shows our two upper bounds for the backward error, $3n^3\varepsilon g_{\text{PP}}$ (or $3n^3\varepsilon g_{\text{CP}}$) and $3n\varepsilon\frac{\||L|\cdot|U|\|_\infty}{\|A\|_\infty}$. It also shows the true backward error, computed as described in Theorem 2.2. Machine epsilon is indicated by a solid horizontal line at $\varepsilon = 2^{-53} \approx 1.1 \cdot 10^{-16}$. Both bounds are indeed bounds on the true backward error but are too large by several order of magnitude. For the Matlab program that produced these plots, see HOMEPAGE/Matlab/pivot.m. ◇

2.4.3. Estimating Condition Numbers

To compute a practical error bound based on a bound like (2.5), we need to estimate $\|A^{-1}\|$. This is also enough to estimate the condition number $\kappa(A) = \|A^{-1}\| \cdot \|A\|$, since $\|A\|$ is easy to compute. One approach is to compute A^{-1} explicitly and compute its norm. However, this would cost $2n^3$, more than the original $\frac{2}{3}n^3$ for Gaussian elimination. (Note that this implies that it is not cheaper to solve $Ax = b$ by computing A^{-1} and then multiplying it by b. This is true even if one has many different b vectors. See Question 2.2.) It is a fact that most users will not bother to compute error bounds if they are expensive.

So instead of computing A^{-1} we will devise a much cheaper algorithm to *estimate* $\|A^{-1}\|$. Such an algorithm is called a *condition estimator* and should have the following properties:

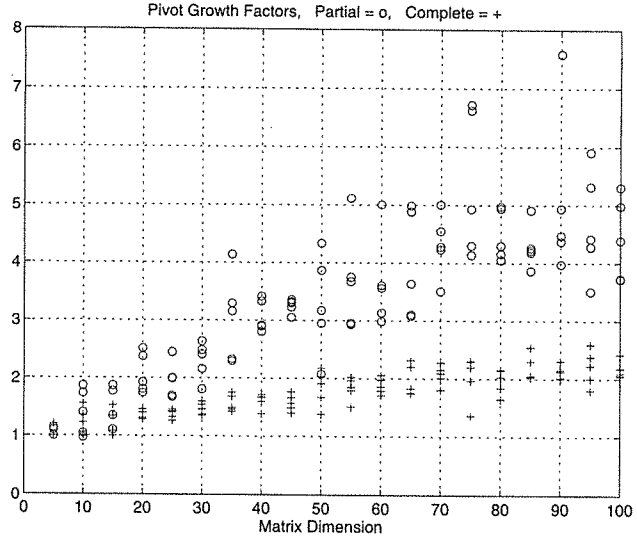


Fig. 2.1. *Pivot growth for random matrices,* $\circ = g_{\text{PP}}$, $+ = g_{\text{CP}}$.

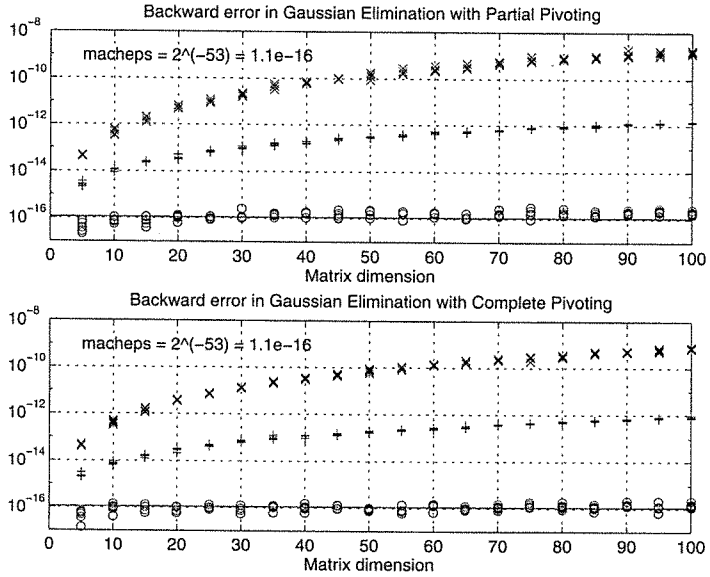


Fig. 2.2. *Backward error in Gaussian elimination on random matrices,* $\times = 3n^3\varepsilon g$, $+ = 3n\||L| \cdot |U|\|_\infty / \|A\|_\infty$, $\circ = \|Ax - b\|_\infty / (\|A\|_\infty \|x\|_\infty)$.

1. Given the L and U factors of A, it should cost $O(n^2)$, which for large enough n is negligible compared to the $\frac{2}{3}n^3$ cost of GEPP.

2. It should provide an estimate which is almost always within a factor of 10 of $\|A^{-1}\|$. This is all one needs for an error bound which tells you about how many decimal digits of accuracy that you have. (A factor-of-10 error is one decimal digit.[11])

There are a variety of such estimators available (see [146] for a survey). We choose to present one that is widely applicable to problems besides solving $Ax = b$, at the cost of being slightly slower than algorithms specialized for $Ax = b$ (but it is still reasonably fast). Our estimator, like most others, is guaranteed to produce only a *lower* bound on $\|A^{-1}\|$, not an upper bound. Empirically, it is almost always within a factor of 10, and usually 2 to 3, of $\|A^{-1}\|$. For the matrices in Figures 2.1 and 2.2, where the condition numbers varied from 10 to 10^5, the estimator equaled the condition number to several decimal places 83% of the time and was .43 times too small at worst. This is more than accurate enough to estimate the number of correct decimal digits in the final answer.

The algorithm estimates the one-norm $\|B\|_1$ of a matrix B, provided that we can compute Bx and B^Ty for arbitrary x and y. We will apply the algorithm to $B = A^{-1}$, so we need to compute $A^{-1}x$ and $A^{-T}y$, i.e., solve linear systems. This costs just $O(n^2)$ given the LU factorization of A. The algorithm was developed in [138, 146, 148], with the latest version in [147]. Recall that $\|B\|_1$ is defined by

$$\|B\|_1 \neq \max_{x\neq 0} \frac{\|Bx\|_1}{\|x\|_1} = \max_j \sum_{i=1}^{n} |b_{ij}|.$$

It is easy for us to show that the maximum over $x \neq 0$ is attained at $x = e_{j_0} = [0, \ldots, 0, 1, 0, \ldots, 0]^T$. (The single nonzero entry is component j_0, where $\max_j \sum_i |b_{ij}|$ occurs at $j = j_0$.)

Searching over all $e_j, j = 1, \ldots, n$, means computing all columns of $B = A^{-1}$; this is too expensive. Instead, since $\|B\|_1 = \max_{\|x\|_1 \leq 1} \|Bx\|_1$, we can use *hill climbing* or *gradient ascent* on $f(x) \equiv \|Bx\|_1$ inside the set $\|x\|_1 \leq 1$. $\|x\|_1 \leq 1$ is clearly a convex set of vectors, and $f(x)$ is a convex function, since $0 \leq \alpha \leq 1$ implies $f(\alpha x + (1-\alpha)y) = \|\alpha Bx + (1-\alpha)By\|_1 \leq \alpha\|Bx\|_1 + (1-\alpha)\|By\|_1 = \alpha f(x) + (1-\alpha)f(y)$.

Doing gradient ascent to maximize $f(x)$ means moving x in the direction of the gradient $\nabla f(x)$ (if it exists) as long as $f(x)$ increases. The convexity of $f(x)$ means $f(y) \geq f(x) + \nabla f(x) \cdot (y - x)$ (if $\nabla f(x)$ exists). To compute ∇f we *assume* all $\sum_j b_{ij}x_j \neq 0$ in $f(x) = \sum_i |\sum_j b_{ij}x_j|$ (this is almost always true). Let $\zeta_i = \text{sign}(\sum_j b_{ij}x_j)$, so $\zeta_i = \pm 1$ and $f(x) = \sum_i \sum_j \zeta_i b_{ij} x_j$. Then $\frac{\partial f}{\partial x_k} = \sum_i \zeta_i b_{ik}$ and $\nabla f = \zeta^T B = (B^T\zeta)^T$.

In summary, to compute $\nabla f(x)$ takes three steps: $w = Bx$, $\zeta = \text{sign}(w)$, and $\nabla f = \zeta^T B$.

[11] As stated earlier, no one has ever found an estimator that approximates $\|A^{-1}\|$ with some guaranteed accuracy and is simultaneously significantly cheaper than explicitly computing A^{-1}. It has been been conjectured that no such estimator exists, but this has not been proven.

ALGORITHM 2.5. *Hager's condition estimator returns a lower bound* $\|w\|_1$ *on* $\|B\|_1$*:*

choose any x *such that* $\|x\|_1 = 1$ /* *e.g.* $x_i = \frac{1}{n}$ */
repeat
  $w = Bx$, $\zeta = \text{sign}(w)$, $z = B^T\zeta$ /* $z^T = \nabla f$ */
  if $\|z\|_\infty \leq z^T x$ *then*
    return $\|w\|_1$
  else
    $x = e_j$ *where* $|z_j| = \|z\|_\infty$
  endif
end repeat

THEOREM 2.6. 1. *When* $\|w\|_1$ *is returned,* $\|w\|_1 = \|Bx\|_1$ *is a local maximum of* $\|Bx\|_1$.

2. *Otherwise,* $\|Be_j\|$ *(at end of loop)* $> \|Bx\|$ *(at start), so the algorithm has made progress in maximizing* $f(x)$.

Proof.

1. In this case, $\|z\|_\infty \leq z^T x$. Near x, $f(x) = \|Bx\|_1 = \sum_i \sum_j \zeta_i b_{ij} x_j$ is linear in x so $f(y) = f(x) + \nabla f(x) \cdot (y - x) = f(x) + z^T(y - x)$, where $z^T = \nabla f(x)$. To show x is a local maximum we want $z^T(y-x) \leq 0$ when $\|y\|_1 = 1$. We compute

$$\begin{aligned} z^T(y-x) &= z^Ty - z^Tx = \sum_i z_i \cdot y_i - z^Tx \leq \sum_i |z_i| \cdot |y_i| - z^Tx \\ &\leq \|z\|_\infty \cdot \|y\|_1 - z^Tx = \|z\|_\infty - z^Tx \leq 0 \quad \text{as desired.} \end{aligned}$$

2. In this case $\|z\|_\infty > z^T x$. Choose $\tilde{x} = e_j \cdot \text{sign}(z_j)$, where j is chosen so that $|z_j| = \|z\|_\infty$. Then

$$\begin{aligned} f(\tilde{x}) &\geq f(x) + \nabla f \cdot (\tilde{x} - x) = f(x) + z^T(\tilde{x} - x) \\ &= f(x) + z^T\tilde{x} - z^Tx = f(x) + |z_j| - z^Tx > f(x), \end{aligned}$$

where the last inequality is true by construction. □

Higham [147, 148] tested a slightly improved version of this algorithm by trying many random matrices of sizes $10, 25, 50$ and condition numbers $\kappa = 10, 10^3, 10^6, 10^9$; in the worst case the computed κ underestimated the

true κ by a factor .44. The algorithm is available in LAPACK as subroutine `slacon`. LAPACK routines like `sgesvx` call `slacon` internally and return the estimated condition number. (They actually return the reciprocal of the estimated condition number, to avoid overflow on exactly singular matrices.) A different condition estimator is available in Matlab as `rcond`. The Matlab routine `cond` computes the exact condition number $\|A^{-1}\|_2\|A\|_2$, using algorithms discussed in section 5.4; it is much more expensive than `rcond`.

Estimating the Relative Condition Number

We can also use the algorithm from the last section to estimate the relative condition number $\kappa_{CR}(A) = \| \, |A^{-1}| \cdot |A| \, \|_\infty$ from bound (2.8) or to evaluate the bound $\| \, |A^{-1}| \cdot |r| \, \|_\infty$ from (2.9). We can reduce both to the same problem, that of estimating $\| \, |A^{-1}| \cdot g \, \|_\infty$, where g is a vector of nonnegative entries. To see why, let e be the vector of all ones. From part 5 of Lemma 1.7, we see that $\|X\|_\infty = \|Xe\|_\infty$ if the matrix X has nonnegative entries. Then

$$\| \, |A^{-1}| \cdot |A| \, \|_\infty = \| \, |A^{-1}| \cdot |A|e \, \|_\infty = \| \, |A^{-1}| \cdot g \, \|_\infty, \quad \text{where} \quad g = |A|e.$$

Here is how we estimate $\| \, |A^{-1}| \cdot g \, \|_\infty$. Let $G = \mathrm{diag}(g_1, \ldots, g_n)$; then $g = Ge$. Thus

$$\begin{aligned} \| \, |A^{-1}| \cdot g \, \|_\infty &= \| \, |A^{-1}| \cdot Ge \, \|_\infty = \| \, |A^{-1}| \cdot G \, \|_\infty = \| \, |A^{-1}G| \, \|_\infty \\ &= \|A^{-1}G\|_\infty. \end{aligned} \tag{2.12}$$

The last equality is true because $\|Y\|_\infty = \| \, |Y| \, \|_\infty$ for any matrix Y. Thus, it suffices to estimate the infinity norm of the matrix $A^{-1}G$. We can do this by applying Hager's algorithm, Algorithm 2.5, to the matrix $(A^{-1}G)^T = GA^{-T}$, to estimate $\|(A^{-1}G)^T\|_1 = \|A^{-1}G\|_\infty$ (see part 6 of Lemma 1.7). This requires us to multiply by the matrix GA^{-T} and its transpose $A^{-1}G$. Multiplying by G is easy since it is diagonal, and we multiply by A^{-1} and A^{-T} using the LU factorization of A, as we did in the last section.

2.4.4. Practical Error Bounds

We present two practical error bounds for our approximate solution $\hat{x}$ of $Ax = b$. For the first bound we use inequality (2.5) to get

$$\text{error} = \frac{\|\hat{x} - x\|_\infty}{\|\hat{x}\|_\infty} \le \|A^{-1}\|_\infty \cdot \frac{\|r\|_\infty}{\|\hat{x}\|_\infty}, \tag{2.13}$$

where $r = A\hat{x} - b$ is the residual. We estimate $\|A^{-1}\|_\infty$ by applying Algorithm 2.5 to $B = A^{-T}$, estimating $\|B\|_1 = \|A^{-T}\|_1 = \|A^{-1}\|_\infty$ (see parts 5 and 6 of Lemma 1.7).

Our second error bound comes from the tighter inequality (2.9):

$$\text{error} = \frac{\|\hat{x} - x\|_\infty}{\|\hat{x}\|_\infty} \le \frac{\| \, |A^{-1}| \cdot |r| \, \|_\infty}{\|\hat{x}\|_\infty}. \tag{2.14}$$

We estimate $\| \, |A^{-1}| \cdot |r| \, \|_\infty$ using the algorithm based on equation (2.12). Error bound (2.14) (modified as described below in the subsection "What can go wrong") is computed by LAPACK routines like `sgesvx`. The LAPACK variable name for the error bound is `FERR`, for Forward ERRor.

EXAMPLE 2.4. We have computed the first error bound (2.13) and the true error for the same set of examples as in Figures 2.1 and 2.2, plotting the result in Figure 2.3. For each problem $Ax = b$ solved with GEPP we plot a $\circ$ at the point (true error, error bound), and for each problem $Ax = b$ solved with GECP we plot a $+$ at the point (true error, error bound). If the error bound were equal to the true error, the $\circ$ or $+$ would lie on the solid diagonal line. Since the error bound always exceeds the true error, the $\circ$s and $+$s lie above this diagonal. When the error bound is less than 10 times larger than the true error, the $\circ$ or $+$ appears between the solid diagonal line and the first superdiagonal dashed line. When the error bound is between 10 and 100 times larger than the true error, the $\circ$ or $+$ appears between the first two superdiagonal dashed lines. Most error bounds are in this range, with a few error bounds as large as 1000 times the true error. Thus, our computed error bound underestimates the number of correct decimal digits in the answer by one or two and in rare cases by as much as three. The Matlab code for producing these graphs is the same as before, HOMEPAGE/Matlab/pivot.m. $\diamond$

EXAMPLE 2.5. We present an example chosen to illustrate the difference between the two error bounds (2.13) and (2.14). This example will also show that GECP can sometimes be more accurate than GEPP. We choose a set of badly scaled examples constructed as follows. Each test matrix is of the form $A = DB$, with the dimension running from 5 to 100. B is equal to an identity matrix plus very small random offdiagonal entries, around 10^{-7}, so it is very well-conditioned. D is a diagonal matrix with entries scaled geometrically from 1 up to 10^{14}. (In other words, $d_{i+1,i+1}/d_{i,i}$ is the same for all i.) The A matrices have condition numbers $\kappa(A) = \|A^{-1}\|_\infty \cdot \|A\|_\infty$ nearly equal to 10^{14}, which is very ill-conditioned, although their relative condition numbers $\kappa_{CR}(A) = \| \, |A^{-1}| \cdot |A| \, \|_\infty = \| \, |B^{-1}| \cdot |B| \, \|_\infty$ are all nearly 1. As before, machine precision is $\varepsilon = 2^{-53} \approx 10^{-16}$. The examples were computed using the same Matlab code HOMEPAGE/Matlab/pivot.m.

The pivot growth factors g_{PP} and g_{CP} were never larger than about 1.33 for any example, and the backward error from Theorem 2.2 never exceeded 10^{-15} in any case. Hager's estimator was very accurate in all cases, returning the true condition number 10^{14} to many decimal places.

Figure 2.4 plots the error bounds (2.13) and (2.14) for these examples, along with the componentwise relative backward error, as given by the formula in Theorem 2.3. The cluster of plus signs in the upper left corner of Figure 2.4(a) shows that while GECP computes the answer with a tiny error near 10^{-15}, the error bound (2.13) is usually closer to 10^{-2}, which is very pessimistic. This

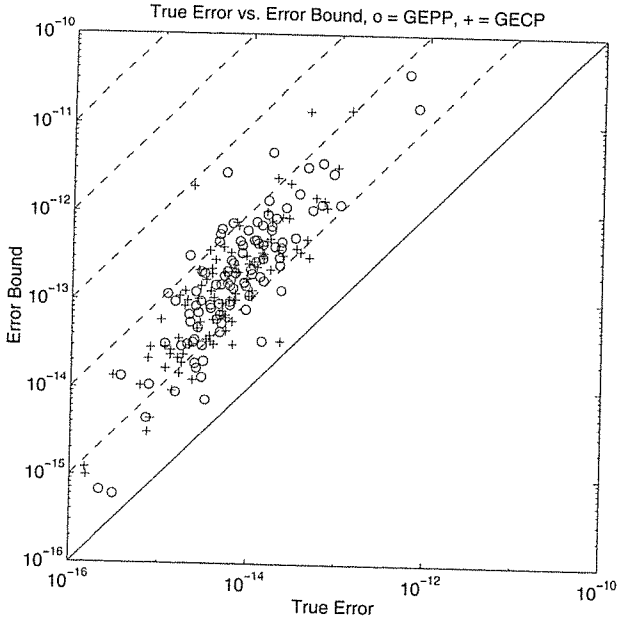
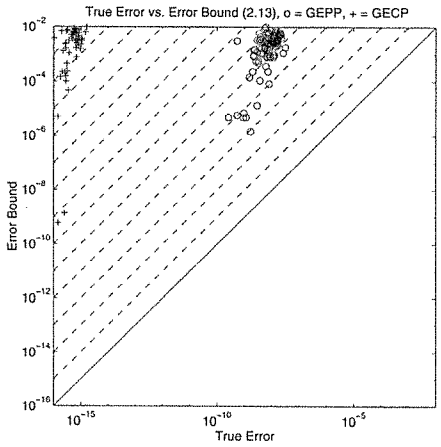


Fig. 2.3. *Error bound* (2.13) *plotted versus true error,* $\circ$ = *GEPP,* $+$ = *GECP.*

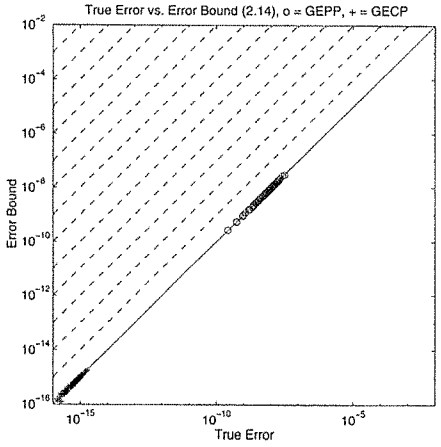
is because the condition number is 10^{14}, and so unless the backward error is much smaller than $\varepsilon \approx 10^{-16}$, which is unlikely, the error bound will be close to $10^{-16}10^{14} = 10^{-2}$. The cluster of circles in the middle top of the same figure shows that GEPP gets a larger error of about 10^{-8}, while the error bound (2.13) is again usually near 10^{-2}.

In contrast, the error bound (2.14) is nearly perfectly accurate, as illustrated by the pluses and circles on the diagonal in Figure 2.4(b). This graph again illustrates that GECP is nearly perfectly accurate, whereas GEPP loses about half the accuracy. This difference in accuracy is explained by Figure 2.4(c), which shows the componentwise relative backward error from Theorem 2.3 for GEPP and GECP. This graph makes it clear that GECP has nearly perfect backward error in the componentwise relative sense, so since the corresponding componentwise relative condition number is 1, the accuracy is perfect. GEPP on the other hand is not completely stable in this sense, losing from 5 to 10 decimal digits.

In section 2.5 we show how to iteratively improve the computed solution $\hat{x}$. One step of this method will make the solution computed by GEPP as accurate as the solution from GECP. Since GECP is significantly more expensive than GEPP in practice, it is very rarely used. $\diamond$

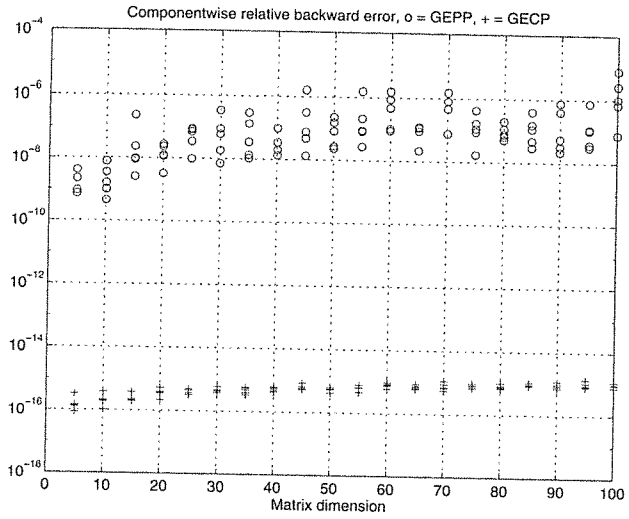


(a)



(b)

Fig. 2.4. (a) *plots the error bound* (2.13) *versus the true error.* (b) *plots the error bound* (2.14) *versus the true error.*



(c)

Fig. 2.4. *Continued.* (c) *plots the componentwise relative backward error from Theorem* 2.3.

What Can Go Wrong

Unfortunately, as mentioned in the beginning of section 2.4, error bounds (2.13) and (2.14) are *not* guaranteed to provide tight bounds in all cases when implemented in practice. In this section we describe the (rare!) ways they can fail, and the partial remedies used in practice.

First, as described in section 2.4.3, the estimate of $\|A^{-1}\|$ from Algorithm 2.5 (or similar algorithms) provides only a lower bound, although the probability is very low that it is more than 10 times too small.

Second, there is a small but nonnegligible probability that roundoff in the evaluation of $r = A\hat{x} - b$ might make $\|r\|$ artificially small, in fact zero, and so also make our computed error bound too small. To take this possibility into account, one can add a small quantity to $|r|$ to account for it: From Question 1.10 we know that the roundoff in evaluating r is bounded by

$$|(A\hat{x} - b) - \mathrm{fl}(A\hat{x} - b)| \le (n+1)\varepsilon(|A| \cdot |\hat{x}| + |b|), \tag{2.15}$$

so we can replace $|r|$ with $|r| + (n+1)\varepsilon(|A| \cdot |\hat{x}| + |b|)$ in bound (2.14) (this is done in the LAPACK code `sgesvx`) or $\|r\|$ with $\|r\| + (n+1)\varepsilon(\|A\| \cdot \|\hat{x}\| + \|b\|)$ in bound (2.13). The factor $n+1$ is usually much too large and can be omitted if desired.

Third, roundoff in performing Gaussian elimination on very ill-conditioned matrices can yield such inaccurate L and U that bound (2.14) is much too low.

EXAMPLE 2.6. We present an example, discovered by W. Kahan, that illustrates the difficulties in getting truly guaranteed error bounds. In this example the matrix A will be *exactly* singular. Therefore any error bound on $\frac{\|x-\hat{x}\|}{\|x\|}$ should be one or larger to indicate that no digits in the computed solution are correct, since the true solution does not exist.

Roundoff error during Gaussian elimination will yield nonsingular but very ill-conditioned factors L and U. With this example, computing using Matlab with IEEE double precision arithmetic, the computed residual r turns out to be *exactly* zero because of roundoff, so both error bounds (2.13) and (2.14) return zero. If we repair bound (2.13) by adding $4\varepsilon(\|A\| \cdot \|\hat{x}\| + \|b\|)$, it will be larger than 1 as desired.

Unfortunately our second, "tighter" error bound (2.14) is about 10^{-7}, erroneously indicating that seven digits of the computed solution are correct.

Here is how the example is constructed. Let $\chi = 3/2^{29}$, $\zeta = 2^{14}$,

$$\begin{aligned} A &= \begin{bmatrix} \chi\cdot\zeta & -\zeta & \zeta \\ \zeta^{-1} & \zeta^{-1} & 0 \\ \zeta^{-1} & -\chi\cdot\zeta^{-1} & \zeta^{-1} \end{bmatrix} \\ &\approx \begin{bmatrix} 9.1553\cdot 10^{-5} & -1.6384\cdot 10^{4} & 1.6384\cdot 10^{4} \\ 6.1035\cdot 10^{-5} & 6.1035\cdot 10^{-5} & 0 \\ 6.1035\cdot 10^{-5} & -3.4106\cdot 10^{-13} & 6.1035\cdot 10^{-5} \end{bmatrix}, \end{aligned}$$

and $b = A \cdot [1, 1+\varepsilon, 1]^T$. A can be computed without any roundoff error, but b has a bit of roundoff, which means that it is not exactly in the space spanned by the columns of A, so $Ax = b$ has no solution. Performing Gaussian elimination, we get

$$L \approx \begin{bmatrix} 1 & 0 & 0 \\ .66666 & 1 & 0 \\ .66666 & 1.0000 & 1 \end{bmatrix}$$

and

$$U \approx \begin{bmatrix} 9.1553\cdot 10^{-5} & -1.6384\cdot 10^{4} & 1.6384\cdot 10^{4} \\ 0 & 1.0923\cdot 10^{4} & -1.0923\cdot 10^{4} \\ 0 & 0 & 1.8190\cdot 10^{-12} \end{bmatrix},$$

yielding a computed value of

$$A^{-1} \approx \begin{bmatrix} 2.0480\cdot 10^{3} & 5.4976\cdot 10^{11} & -5.4976\cdot 10^{11} \\ -2.0480\cdot 10^{3} & -5.4976\cdot 10^{11} & 5.4976\cdot 10^{11} \\ -2.0480\cdot 10^{3} & -5.4976\cdot 10^{11} & 5.4976\cdot 10^{11} \end{bmatrix}.$$

This means the computed value of $|A^{-1}| \cdot |A|$ has all entries approximately equal to $6.7109 \cdot 10^7$, so $\kappa_{CR}(A)$ is computed to be $O(10^7)$. In other words, the

error bound indicates that about $16 - 7 = 9$ digits of the computed solution are accurate, whereas none are.

Barring large pivot growth, one can prove that bound (2.13) (with $\|r\|$ appropriately increased) cannot be made artificially small by the phenomenon illustrated here.

- Similarly, Kahan has found a family of n-by-n singular matrices, where changing one tiny entry (about 2^{-n}) to zero lowers $\kappa_{CR}(A)$ to $O(n^3)$. One could similarly construct examples where A was not exactly singular, so that bounds (2.13) and (2.14) were correct in exact arithmetic, but where roundoff made them much too small. $\diamond$

2.5. Improving the Accuracy of a Solution

We have just seen that the error in solving $Ax = b$ may be as large as $\kappa(A)\varepsilon$. If this error is too large, what can we do? One possibility is to rerun the entire computation in higher precision, but this may be quite expensive in time and space. Fortunately, as long as $\kappa(A)$ is not too large, there are much cheaper methods available for getting a more accurate solution.

To solve any equation $f(x) = 0$, we can try to use Newton's method to improve an approximate solution x_i to get $x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$. Applying this to $f(x) = Ax - b$ yields one step of *iterative refinement*:

$$\begin{aligned} &r = Ax_i - b \\ &\text{solve } Ad = r \text{ for } d \\ &x_{i+1} = x_i - d \end{aligned}$$

If we could compute $r = Ax_i - b$ exactly and solve $Ad = r$ exactly, we would be done in one step, which is what we expect from Newton applied to a linear problem. Roundoff error prevents this immediate convergence. The algorithm is interesting and of use precisely when A is so ill-conditioned that solving $Ad = r$ (and $Ax_0 = b$) is rather inaccurate.

THEOREM 2.7. *Suppose that r is computed in* double *precision and $\kappa(A) \cdot \varepsilon < c \equiv \frac{1}{3n^3g+1} < 1$, where n is the dimension of A and g is the pivot growth factor. Then repeated iterative refinement converges with*

$$\frac{\|x_i - A^{-1}b\|_\infty}{\|A^{-1}b\|_\infty} = O(\varepsilon).$$

Note that the condition number does not appear in the final error bound. This means that we compute the answer accurately independent *of the condition number, provided that $\kappa(A)\varepsilon$ is sufficiently less than* 1. *(In practice, c is too conservative an upper bound, and the algorithm often succeeds even when $\kappa(A)\varepsilon$ is greater than c.)*

Sketch of Proof. In order to keep the proof transparent, we will take only the most important rounding errors into account. For brevity, we abbreviate $\|\cdot\|_\infty$ by $\|\cdot\|$. Our goal is to show that

$$\|x_{i+1} - x\| \le \frac{\kappa(A)\varepsilon}{c}\|x_i - x\| \equiv \zeta\|x_i - x\|.$$

By assumption, $\zeta < 1$, so this inequality implies that the error $\|x_{i+1} - x\|$ decreases monotonically to zero. (In practice it will not decrease all the way to zero because of rounding error in the assignment $x_{i+1} = x_i - d$, which we are ignoring.)

We begin by estimating the error in the computed residual r. We get $r = \mathrm{fl}(Ax_i - b) = Ax_i - b + f$, where by the result of Question 1.10 $|f| \le n\varepsilon^2(|A| \cdot |x_i| + |b|) + \varepsilon|Ax_i - b| \approx \varepsilon|Ax_i - b|$. The ε^2 term comes from the double precision computation of r, and the ε term comes from rounding the double precision result back to single precision. Since $\varepsilon^2 \ll \varepsilon$, we will neglect the ε^2 term in the bound on $|f|$.

Next we get $(A+\delta A)d = r$, where from bound (2.11) we know that $\|\delta A\| \le \gamma \cdot \varepsilon \cdot \|A\|$, where $\gamma = 3n^3g$, although this is usually much too large. As mentioned earlier, we simplify matters by assuming $x_{i+1} = x_i - d$ exactly.

Continuing to ignore all ε^2 terms, we get

$$\begin{aligned} d &= (A + \delta A)^{-1}r = (I + A^{-1}\delta A)^{-1}A^{-1}r \\ &= (I + A^{-1}\delta A)^{-1}A^{-1}(Ax_i - b + f) \\ &= (I + A^{-1}\delta A)^{-1}(x_i - x + A^{-1}f) \\ &\approx (I - A^{-1}\delta A)(x_i - x + A^{-1}f) \\ &\approx x_i - x - A^{-1}\delta A(x_i - x) + A^{-1}f. \end{aligned}$$

Therefore $x_{i+1} - x = x_i - d - x = A^{-1}\delta A(x_i - x) - A^{-1}f$ and so

$$\begin{aligned} \|x_{i+1} - x\| &\le \|A^{-1}\delta A(x_i - x)\| + \|A^{-1}f\| \\ &\le \|A^{-1}\| \cdot \|\delta A\| \cdot \|x_i - x\| + \|A^{-1}\| \cdot \varepsilon \cdot \|Ax_i - b\| \\ &\le \|A^{-1}\| \cdot \|\delta A\| \cdot \|x_i - x\| + \|A^{-1}\| \cdot \varepsilon \cdot \|A(x_i - x)\| \\ &\le \|A^{-1}\| \cdot \gamma\varepsilon \cdot \|A\| \cdot \|x_i - x\| \\ &\quad + \|A^{-1}\| \cdot \|A\| \cdot \varepsilon \cdot \|x_i - x\| \\ &= \|A^{-1}\| \cdot \|A\| \cdot \varepsilon \cdot (\gamma + 1) \cdot \|x_i - x\|, \end{aligned}$$

so if

$$\zeta = \|A^{-1}\| \cdot \|A\| \cdot \varepsilon(\gamma + 1) = \kappa(A)\varepsilon/c < 1,$$

then we have convergence. $\square$

Iterative refinement (or other variations of Newton's method) can be used to improve accuracy for many other problems of linear algebra as well.

2.5.1. Single Precision Iterative Refinement

This section may be skipped on a first reading.

Sometimes double precision is not available to run iterative refinement. For example, if the input data is already in double precision, we would need to compute the residual r in *quadruple* precision, which may not be available. On some machines, such as the Intel Pentium, double-extended precision is available, which provides 11 more bits of fraction than double precision (see section 1.5). This is not as accurate as quadruple precision (which would need at least $2 \cdot 53 = 106$ fraction bits) but still improves the accuracy noticeably.

But if none of these options are available, one could still run iterative refinement while computing the residual r in single precision (i.e., the same precision as the input data). In this case, Theorem 2.7 does not hold any more. On the other hand, the following theorem shows that under certain technical assumptions, one step of iterative refinement in single precision is still worth doing because it reduces the componentwise relative backward error as defined in Theorem 2.3 to $O(\varepsilon)$. If the corresponding relative condition number $\kappa_{CR}(A) = \| |A^{-1}| \cdot |A| \|_\infty$ from section 2.2.1 is significantly smaller than the usual condition number $\kappa(A) = \|A^{-1}\|_\infty \cdot \|A\|_\infty$, then the answer will also be more accurate.

THEOREM 2.8. *Suppose that r is computed in single precision and*

$$\|A^{-1}\|_\infty \cdot \|A\|_\infty \cdot \frac{\max_i(|A| \cdot |x|)_i}{\min_i(|A| \cdot |x|)_i} \cdot \varepsilon < 1.$$

Then one step of iterative refinement yields x_1 such that $(A + \delta A)x_1 = b + \delta b$ with $|\delta a_{ij}| = O(\varepsilon)|a_{ij}|$ and $|\delta b_i| = O(\varepsilon)|b_i|$. In other words, the componentwise relative backward error is as small as possible. For example, this means that if A and b are sparse, then δA and δb have the same sparsity structures as A and b, respectively.

For a proof, see [149] as well as [14, 225, 226, 227] for more details.

Single precision iterative refinement and the error bound (2.14) are implemented in LAPACK routines like `sgesvx`.

EXAMPLE 2.7. We consider the same matrices as in Example 2.5 and perform one step of iterative refinement in the same precision as the rest of the computation ($\varepsilon \approx 10^{-16}$). For these examples, the usual condition number is $\kappa(A) \approx 10^{14}$, whereas $\kappa_{CR}(A) \approx 1$, so we expect a large accuracy improvement. Indeed, the componentwise relative error for GEPP is driven below 10^{-15}, and the corresponding error from (2.14) is driven below 10^{-15} as well. The Matlab code for this example is HOMEPAGE/Matlab/pivot.m. ◇

2.5.2. Equilibration

There is one more common technique for improving the error in solving a linear system: *equilibration*. This refers to choosing an appropriate diagonal matrix D and solving $DAx = Db$ instead of $Ax = b$. D is chosen to try to make the condition number of DA smaller than that of A. In Example 2.7 for instance, choosing d_{ii} to be the reciprocal of the two-norm of row i of A would make DA nearly equal to the identity matrix, reducing its condition number from 10^{14} to 1. It is possible to show that choosing D this way reduces the condition number of DA to within a factor of $\sqrt{n}$ of its smallest possible value for any diagonal D [244]. In practice we may also choose two diagonal matrices D_{row} and D_{col} and solve $(D_{row}AD_{col})\bar{x} = D_{row}b$, $x = D_{col}\bar{x}$.

The techniques of iterative refinement and equilibration are implemented in the LAPACK subroutines like `sgerfs` and `sgeequ`, respectively. These are in turn used by driver routines like `sgesvx`.

2.6. Blocking Algorithms for Higher Performance

At the end of section 2.3, we said that changing the order of the three nested loops in the implementation of Gaussian elimination in Algorithm 2.2 could change the execution speed by orders of magnitude, depending on the computer and the problem being solved. In this section we will explore why this is the case and describe some carefully written linear algebra software which takes these matters into account. These implementations use so-called *block algorithms*, because they operate on square or rectangular subblocks of matrices in their innermost loops rather than on entire rows or columns. These codes are available in public-domain software libraries such as LAPACK (in Fortran, at NETLIB/lapack)[12] and ScaLAPACK (at NETLIB/scalapack). LAPACK (and its versions in other languages) are suitable for PCs, workstations, vector computers, and shared-memory parallel computers. These include the Sun SPARCcenter 2000 [238], SGI Power Challenge [223], DEC AlphaServer 8400 [103], and Cray C90/J90 [253, 254]. ScaLAPACK is suitable for distributed-memory parallel computers, such as the IBM SP-2 [256], Intel Paragon [257], Cray T3 series [255], and networks of workstations [9]. These libraries are available on NETLIB, including comprehensive manuals [10, 34].

A more comprehensive discussion of algorithms for high performance (especially parallel) machines may be found on the World Wide Web at PARALLEL_HOMEPAGE.

LAPACK was originally motivated by the poor performance of its predecessors LINPACK and EISPACK (also available on NETLIB) on some high-performance machines. For example, consider the table below, which presents the speed in Mflops of LINPACK's Cholesky routine `spofa` on a Cray YMP, a supercomputer of the late 1980s. Cholesky is a variant of Gaussian elimination suitable for symmetric positive definite matrices. It is discussed in depth in

[12] A C translation of LAPACK, called CLAPACK (at NETLIB/clapack), is also available. LAPACK++ (at NETLIB/c++/lapack++)) and LAPACK90 (at NETLIB/lapack90)) are C++ and Fortran 90 interfaces to LAPACK, respectively.

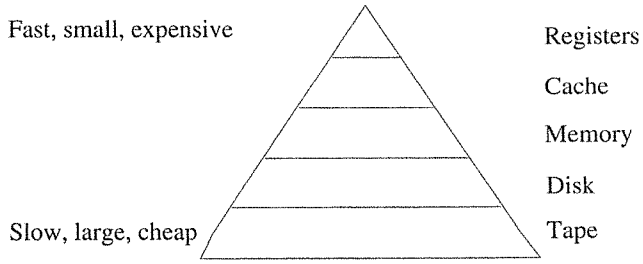
section 2.7; here it suffices to know that it is very similar to Algorithm 2.2. The table also includes the speed of several other linear algebra operations. The Cray YMP is a parallel computer with up to 8 processors that can be used simultaneously, so we include one column of data for 1 processor and another column where all 8 processors are used.

	1 Proc.	8 Procs.
Maximum speed	330	2640
Matrix-matrix multiply ($n = 500$)	312	2425
Matrix-vector multiply ($n = 500$)	311	2285
Solve $TX = B$ ($n = 500$)	309	2398
Solve $Tx = b$ ($n = 500$)	272	584
LINPACK (Cholesky, $n = 500$)	72	72
LAPACK (Cholesky, $n = 500$)	290	1414
LAPACK (Cholesky, $n = 1000$)	301	2115

The top line, the maximum speed of the machine, is an upper bound on the numbers that follow. The basic linear algebra operations on the next four lines have been measured using subroutines especially designed for high speed on the Cray YMP. They all get reasonably close to the maximum possible speed, except for solving $Tx = b$, a single triangular system of linear equations, which does not use 8 processors effectively. Solving $TX = B$ refers to solving triangular systems with many right-hand sides (B is a square matrix). These numbers are for large matrices and vectors ($n = 500$).

The Cholesky routine from LINPACK in the sixth line of the table executes significantly more slowly than these other operations, even though it is working on as large a matrix as the previous operations and doing mathematically similar operations. This poor performance leads us to try to reorganize Cholesky and other linear algebra routines to go as fast as their simpler counterparts like matrix-matrix multiplication. The speeds of these reorganized codes from LAPACK are given in the last two lines of the table. It is apparent that the LAPACK routines come much closer to the maximum speed of the machine. We emphasize that the LAPACK and LINPACK Cholesky routines perform the same floating operations, but in a different order.

To understand how these speedups were attained, we must understand how the time is spent by the computer while executing. This in turn requires us to understand how computer memories operate. It turns out that all computer memories, from the cheapest personal computer to the biggest supercomputer, are built as *hierarchies*, with a series of different kinds of memories ranging from very fast, expensive, and therefore small memory at the top of the hierarchy down to slow, cheap, and very large memory at the bottom.



For example, registers form the fastest memory, then cache, main memory, disks, and finally tape as the slowest, largest, and cheapest. Useful arithmetic and logical operations can be done *only* on data at the top of the hierarchy, in the registers. Data at one level of the memory hierarchy can move to adjacent levels—for example, moving between main memory and disk. The speed at which data move is high near the top of the hierarchy (between registers and cache) and slow near the bottom (between and disk and main memory). In particular, the speed at which arithmetic is done is much faster than the speed at which data is transferred between lower levels in the memory hierarchy, by factors of 10s or even 10000s, depending on the level. This means that an ill-designed algorithm may spend most of its time moving data from the bottom of the memory hierarchy to the registers in order to perform useful work rather than actually doing the work.

Here is an example of a simple algorithm which unfortunately cannot avoid spending most of its time moving data rather than doing useful arithmetic. Suppose that we want to add two large n-by-n matrices, large enough so that they fit only in a large, slow level of the memory hierarchy. To add them, they must be transferred a piece at a time up to the registers to do the additions, and the sums must be transferred back down. Thus, there are exactly 3 memory transfers between fast and slow memory (reading 2 summands into fast memory and writing 1 sum back to slow memory) for every addition performed. If the time to do a floating point operation is t_{arith} seconds and the time to move a word of data between memory levels is t_{mem} seconds, where $t_{\text{mem}} \gg t_{\text{arith}}$, then the execution time of this algorithm is $n^2(t_{\text{arith}} + 3t_{\text{mem}})$, which is much larger than than the time $n^2 t_{\text{arith}}$ required for the arithmetic alone. This means that matrix addition is doomed to run at the speed of the slowest level of memory in which the matrices reside, rather than the much higher speed of addition. In contrast, we will see later that other operations, such as matrix-matrix multiplication, can be made to run at the speed of the fastest level of the memory, even if the data are originally stored in the slowest.

LINPACK's Cholesky routine runs so slowly because it was *not* designed to minimize memory movement on machines such as the Cray YMP.[13] In contrast, matrix-matrix multiplication and the three other basic linear algebra

[13] It was designed to reduce another kind of memory movement, *page faults* between main memory and disk.

algorithms measured in the table were specialized to minimize data movement on a Cray YMP.

2.6.1. Basic Linear Algebra Subroutines (BLAS)

Since it is not cost-effective to write a special version of every routine like Cholesky for every new computer, we need a more systematic approach. Since operations like matrix-matrix multiplication are so common, computer manufacturers have standardized them as the *Basic Linear Algebra Subroutines*, or *BLAS* [169, 89, 87], and optimized them for their machines. In other words, a library of subroutines for matrix-matrix multiplication, matrix-vector multiplication, and other similar operations is available with a standard Fortran or C interface on high performance machines (and many others), but underneath they have been optimized for each machine. Our goal is to take advantage of these optimized BLAS by reorganizing algorithms like Cholesky so that they call the BLAS to perform most of their work.

In this section we will discuss the BLAS in general. In section 2.6.2, we will describe how to optimize matrix multiplication in particular. Finally, in section 2.6.3, we show how to reorganize Gaussian elimination so that most of its work is performed using matrix multiplication.

Let us examine the BLAS more carefully. Table 2.1 counts the number of memory references and floating points operations performed by three related BLAS. For example, the number of memory references needed to implement the `saxpy` operation in line 1 of the table is $3n + 1$, because we need to read n values of x_i, n values of y_i, and 1 value of α from slow memory to registers, and then write n values of y_i back to slow memory. The last column gives the ratio q of flops to memory references (its highest-order term in n only).

The significance of q is that it tells us roughly how many flops that we can perform per memory reference or how much useful work we can do compared to the time moving data. This tells us how fast the algorithm can *potentially* run. For example, suppose that an algorithm performs f floating points operations, each of which takes t_{arith} seconds, and m memory references, each of which takes t_{mem} seconds. Then the total running time is as large as

$$f \cdot t_{\text{arith}} + m \cdot t_{\text{mem}} = f \cdot t_{\text{arith}} \cdot \left(1 + \frac{m}{f} \frac{t_{\text{mem}}}{t_{\text{arith}}}\right) = f \cdot t_{\text{arith}} \cdot \left(1 + \frac{1}{q} \frac{t_{\text{mem}}}{t_{\text{arith}}}\right),$$

assuming that the arithmetic and memory references are not performed in parallel. Therefore, the larger the value of q, the closer the running time is to the best possible running time $f \cdot t_{\text{arith}}$, which is how long the algorithm would take if all data were in registers. This means that algorithms with the larger q values are better building blocks for other algorithms.

Table 2.1 reflects a hierarchy of operations: Operations such as `saxpy` perform $O(n^1)$ flops on vectors and offer the worst q values; these are called Level 1 BLAS, or BLAS1 [169], and include inner products, multiplying a

Operation	Definition	f	m	$q = f/m$
`saxpy` (BLAS1)	$y = \alpha \cdot x + y$ or $y_i = \alpha x_i + y_i$ $i = 1, \ldots, n$	$2n$	$3n + 1$	2/3
Matrix-vector mult (BLAS2)	$y = A \cdot x + y$ or $y_i = \sum_{j=1}^{n} a_{ij} x_j + y_i$ $i = 1, \ldots, n$	$2n^2$	$n^2 + 3n$	2
Matrix-matrix mult (BLAS3)	$C = A \cdot B + C$ or $c_{ij} = \sum_{k=1}^{n} a_{ik} b_{jk} + c_{ij}$ $i, j = 1, \ldots, n$	$2n^3$	$4n^2$	$n/2$

Table 2.1. *Counting floating point operations and memory references for the BLAS. f is the number of floating point operations, and m is the number of memory references.*

scalar times a vector and other simple operations. Operations such as matrix-vector multiplication perform $O(n^2)$ flops on matrices and vectors and offer slightly better q values; these are called Level 2 BLAS, or BLAS2 [89, 88], and include solving triangular systems of equations and rank-1 updates of matrices ($A + xy^T$, x and y column vectors). Operations such as matrix-matrix multiplication perform $O(n^3)$ flops on pairs of matrices and offer the best q values; these are called Level 3 BLAS, or BLAS3 [87, 86], and include solving triangular systems of equations with many right-hand sides.

The directory NETLIB/blas includes documentation and (unoptimized) implementations of all the BLAS. For a quick summary of all the BLAS, see NETLIB/blas/blasqr.ps. This summary also appears in [10, App. C] (or NETLIB/lapack/lug/lapack_lug.html).

Since the Level 3 BLAS have the highest q values, we endeavor to reorganize our algorithms in terms of operations such as matrix-matrix multiplication rather than `saxpy` or matrix-vector multiplication. (LINPACK's Cholesky is constructed in terms of calls to `saxpy`.) We emphasize that such reorganized algorithms will only be faster when using BLAS that have been optimized.

2.6.2. How to Optimize Matrix Multiplication

Let us examine in detail how to implement matrix multiplication $C = A \cdot B + C$ to minimize the number of memory moves and so optimize its performance. We will see that the performance is sensitive to the implementation details. To simplify our discussion, we will use the following machine model. We assume that matrices are stored columnwise, as in Fortran. (It is easy to modify the examples below if matrices are stored rowwise as in C.) We assume that there are two levels of memory hierarchy, fast and slow, where the slow memory is large enough to contain the three $n \times n$ matrices A, B, and C, but the fast memory contains only M words where $2n < M \ll n^2$; this means that

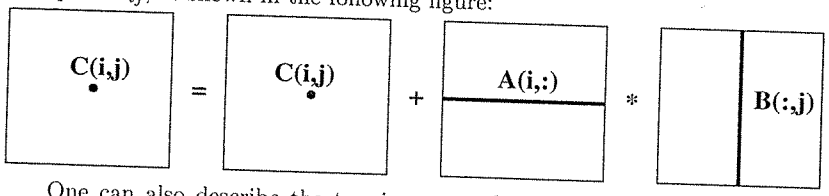
the fast memory is large enough to hold two matrix columns or rows but not a whole matrix. We further assume that the data movement is under programmer control. (In practice, data movement may be done automatically by hardware, such as the cache controller. Nonetheless, the basic optimization scheme remains the same.)

The simplest matrix-multiplication algorithm that one might try consists of three nested loops, which we have annotated to indicate the data movements.

ALGORITHM 2.6. *Unblocked matrix multiplication (annotated to indicate memory activity):*

```
for i = 1 to n
    { Read row i of A into fast memory }
    for j = 1 to n
        { Read C_ij into fast memory }
        { Read column j of B into fast memory }
        for k = 1 to n
            C_ij = C_ij + A_ik · B_kj
        end for
        { Write C_ij back to slow memory }
    end for
end for
```

The innermost loop is doing a dot product of row i of A and column j of B to compute C_{ij}, as shown in the following figure:

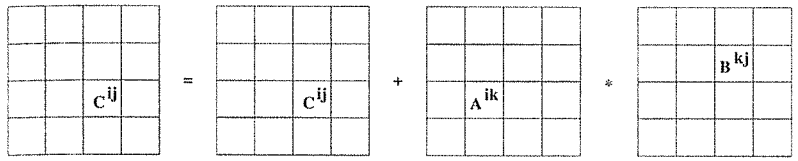


One can also describe the two innermost loops (on j and k) as doing a vector-matrix multiplication of the ith row of A times the matrix B to get the ith row of C. This is a hint that we will not perform any better than these BLAS1 and BLAS2 operations, since they are within the innermost loops.

Here is the detailed count of memory references: n^3 for reading B n times (once for each value of i); n^2 for reading A one row at a time and keeping it in fast memory until it is no longer needed; and $2n^2$ for reading one entry of C at a time, keeping it in fast memory until it is completely computed, and then moving it back to slow memory. This comes to $n^3 + 3n^2$ memory moves, or $q = 2n^3/(n^2+3n^2) \approx 2$, which is no better than the Level 2 BLAS and far from the maximum possible $n/2$ (see Table 2.1). If $M \ll n$, so that we cannot keep a full row of A in fast memory, q further decreases to 1, since the algorithm reduces to a sequence of inner products, which are Level 1 BLAS. For every

permutation of the three loops on i, j, and k, one gets another algorithm with q about the same.

Our preferred algorithm uses *blocking*, where C is broken into an $N \times N$ block matrix with $n/N \times n/N$ blocks C^{ij}, and A and B are similarly partitioned, as shown below for $N = 4$. The algorithm becomes



$$C^{ij} = C^{ij} + \sum_{k=1}^{N} A^{ik} * B^{kj}$$

ALGORITHM 2.7. *Blocked matrix multiplication (annotated to indicate memory activity):*

```
for i = 1 to N
    for j = 1 to N
    { Read C^ij into fast memory }
        for k = 1 to N
            { Read A^ik into fast memory }
            { Read B^kj into fast memory }
            C^ij = C^ij + A^ik · B^kj
        end for
    { Write C^ij back to slow memory }
    end for
end for
```

Our memory reference count is as follows: $2n^2$ for reading and writing each block of C once, Nn^2 for reading A N times (reading each n/N-by-n/N submatrix A^{ik} N^3 times), and Nn^2 for reading B N times (reading each n/N-by-n/N submatrix B^{kj} N^3 times), for a total of $(2N + 2)n^2 \approx 2Nn^2$ memory references. So we want to choose N as small as possible to minimize the number of memory references. But N is subject to the constraint $M \geq 3(n/N)^2$, which means that one block each from A, B, and C must fit in fast memory simultaneously. This yields $N \approx n\sqrt{3/M}$, and so $q \approx (2n^3)/(2Nn^2) \approx \sqrt{M/3}$, which is much better than the previous algorithm. In particular q grows independently of n as M grows, which means that we expect the algorithm to be fast for any matrix size n and to go faster if the fast memory size M is increased. These are both attractive properties.

In fact, it can be shown that Algorithm 2.7 is asymptotically optimal [151]. In other words, no reorganization of matrix-matrix multiplication (that performs the same $2n^3$ arithmetic operations) can have a q larger than $O(\sqrt{M})$.

On the other hand, this brief analysis ignores a number of practical issues:

1. A real code will have to deal with nonsquare matrices, for which the optimal block sizes may not be square.
2. The cache and register structure of a machine will strongly affect the best shapes of submatrices.
3. There may be special hardware instructions that perform both a multiplication and an addition in one cycle. It may also be possible to execute several multiply-add operations simultaneously if they do not interfere.

For a detailed discussion of these issues for one high-performance workstation, the IBM RS6000/590, see [1], PARALLEL_HOMEPAGE, or http://www.rs6000.ibm.com/resource/technology/essl.html. Figure 2.5 shows the speeds of the three basic BLAS for this machine. The horizontal axis is matrix size, and the vertical axis is speed in Mflops. The peak machine speed is 266 Mflops. The top curve (peaking near 250 Mflops) is square matrix-matrix multiplication. The middle curve (peaking near 100 Mflops) is square matrix-vector multiplication, and the bottom curve (peaking near 75 Mflops) is saxpy. Note that the speed increases for larger matrices. This is a common phenomenon and means that we will try to develop algorithms whose internal matrix-multiplications use as large matrices as reasonable.

Both the above matrix-matrix multiplication algorithms perform $2n^3$ arithmetic operations. It turns out that there are other implementations of matrix-matrix multiplication that use far fewer operations. Strassen's method [3] was the first of these algorithms to be discovered and is the simplest to explain. This algorithm multiplies matrices recursively by dividing them into 2×2 block matrices and multiplying the subblocks using seven matrix multiplications (recursively) and 18 matrix additions of half the size; this leads to an asymptotic complexity of $n^{\log_2 7} \approx n^{2.81}$ instead of n^3.

ALGORITHM 2.8. *Strassen's matrix multiplication algorithm:*

C = *Strassen(A,B,n)*
  /* *Return* $C = A * B$, *where* A *and* B *are* n*-by-*n*;*
    Assume n *is a power of* 2 */
  if $n = 1$
    return $C = A * B$ /* *scalar multiplication* */
  else

$$\textit{Partition } A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \textit{ and } B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

where the subblocks A_{ij} *and* B_{ij} *are* $n/2$*-by-*$n/2$

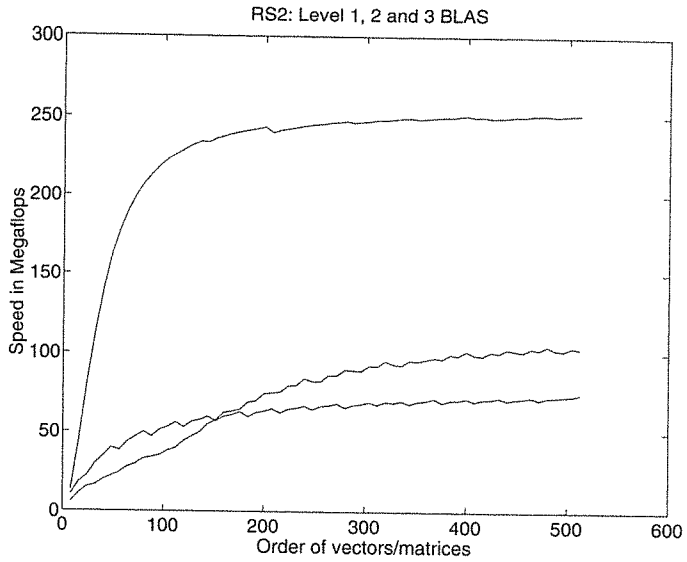


Fig. 2.5. *BLAS speed on the IBM RS* 6000/590.

$$\begin{aligned}
P_1 &= \textit{Strassen}(\, A_{12} - A_{22},\ B_{21} + B_{22},\ n/2\,) \\
P_2 &= \textit{Strassen}(\, A_{11} + A_{22},\ B_{11} + B_{22},\ n/2\,) \\
P_3 &= \textit{Strassen}(\, A_{11} - A_{21},\ B_{11} + B_{12},\ n/2\,) \\
P_4 &= \textit{Strassen}(\, A_{11} + A_{12},\ B_{22},\ n/2\,) \\
P_5 &= \textit{Strassen}(\, A_{11},\ B_{12} - B_{22},\ n/2\,) \\
P_6 &= \textit{Strassen}(\, A_{22},\ B_{21} - B_{11},\ n/2\,) \\
P_7 &= \textit{Strassen}(\, A_{21} + A_{22},\ B_{11},\ n/2\,) \\
C_{11} &= P_1 + P_2 - P_4 + P_6 \\
C_{12} &= P_4 + P_5 \\
C_{21} &= P_6 + P_7 \\
C_{22} &= P_2 - P_3 + P_5 - P_7
\end{aligned}$$

  return $C = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$
end if

It is tedious but straightforward to confirm by induction that this algorithm multiplies matrices correctly (see Question 2.21). To show that its complexity is $O(n^{\log_2 7})$, we let $T(n)$ be the number of additions, subtractions, and multiplications performed by the algorithm. Since the algorithm performs seven recursive calls on matrices of size $n/2$, and 18 additions of $n/2$-by-$n/2$ matrices, we can write down the recurrence $T(n) = 7T(n/2)+18(n/2)^2$. Changing variables

from n to $m = \log_2 n$, we get a new recurrence $\bar{T}(m) = 7\bar{T}(m-1) + 18(2^{m-1})^2$, where $\bar{T}(m) = T(2^m)$. We can confirm that this linear recurrence for $\bar{T}$ has a solution $\bar{T}(m) = O(7^m) = O(n^{\log_2 7})$.

The value of Strassen's algorithm is not just this asymptotic complexity but its reduction of the problem to smaller subproblems which eventually fit in fast memory; once the subproblems fit in fast memory, standard matrix multiplication may be used. This approach has led to speedups on relatively large matrices on some machines [22]. A drawback is the need for significant workspace and somewhat lower numerical stability, although it is adequate for many purposes [77]. There are a number of other even faster matrix multiplication algorithms; the current record is about $O(n^{2.376})$, due to Winograd and Coppersmith [263]. But these algorithms only perform fewer operations than Strassen for impractically large values of n. For a survey see [195].

2.6.3. Reorganizing Gaussian Elimination to Use Level 3 BLAS

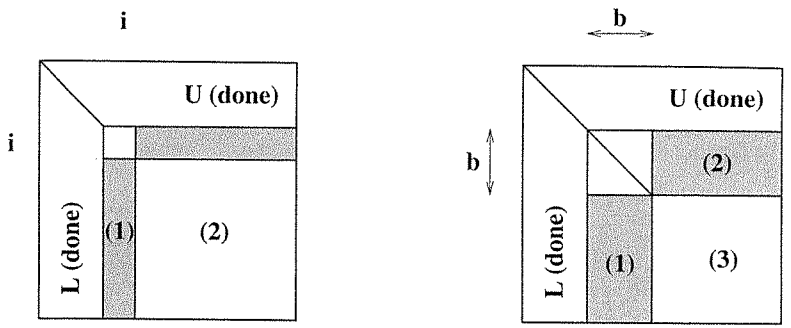
We will reorganize Gaussian elimination to use, first, the Level 2 BLAS and, then, the Level 3 BLAS. For simplicity, we assume that no pivoting is necessary.

Indeed, Algorithm 2.4 is already a Level 2 BLAS algorithm, because most of the work is done in the second line, $A(i+1:n, i+1:n) = A(i+1:n, i+1:n) - A(i+1:n, i) * A(i, i+1:n)$, which is a *rank-1 update* of the submatrix $A(i+1:n, i+1:n)$. The other arithmetic in the algorithm, $A(i+1:n, i) = A(i+1:n, i)/A(i, i)$, is actually done by multiplying the vector $A(i+1:n, i)$ by the scalar $1/A(i, i)$, since multiplication is much faster than division; this is also a Level 1 BLAS operation. We need to modify Algorithm 2.4 slightly because we will use it within the Level 3 version.

ALGORITHM 2.9. *Level 2 BLAS implementation of LU factorization without pivoting for an m-by-n matrix A, where $m \geq n$: Overwrite A by the m-by-n matrix L and m-by-m matrix U. We have numbered the important lines for later reference.*

for $i = 1$ *to* $\min(m-1, n)$
(1) $\quad A(i+1:m, i) = A(i+1:m, i)/A(i, i)$
$\quad$ *if* $i < n$
(2) $\quad\quad A(i+1:m, i+1:n) = A(i+1:m, i+1:n) -$
$\quad\quad\quad A(i+1:m, i) \cdot A(i, i+1:n)$
end for

The left side of Figure 2.6 illustrates Algorithm 2.9 applied to a square matrix. At step i of the algorithm, columns 1 to $i-1$ of L and rows 1 to $i-1$ of U are already done, column i of L and row i of U are to be computed, and the trailing submatrix of A is to be updated by a rank-1 update. On the left side of Figure 2.6, the submatrices are labeled by the lines of the algorithm ((1) or (2)) that update them. The rank-1 update in line (2) is to subtract the



Step i of Level 2 BLAS Implementation of LU

Step i of Level 3 BLAS Implementation of LU

Fig. 2.6. *Level 2 and Level 3 BLAS implementations of LU factorization.*

product of the shaded column and the shaded row from the submatrix labeled (2).

The Level 3 BLAS algorithm will reorganize this computation by *delaying the update* of submatrix (2) for b steps, where b is a small integer called the *block size*, and later applying b rank-1 updates all at once in a single matrix-matrix multiplication. To see how to do this, suppose that we have already computed the first $i-1$ columns of L and rows of U, yielding

$$A = \begin{array}{c} \\ i-1 \\ b \\ n-b-i+1 \end{array} \begin{array}{c} \begin{array}{ccc} i-1 & b & n-b-i+1 \end{array} \\ \begin{pmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{pmatrix} \end{array}$$
$$= \begin{bmatrix} L_{11} & 0 & 0 \\ L_{21} & I & 0 \\ L_{31} & 0 & I \end{bmatrix} \cdot \begin{bmatrix} U_{11} & U_{21} & U_{31} \\ 0 & \tilde{A}_{22} & \tilde{A}_{23} \\ 0 & \tilde{A}_{32} & \tilde{A}_{33} \end{bmatrix},$$

where all the matrices are partitioned the same way. This is shown on the right side of Figure 2.6. Now apply Algorithm 2.9 to the submatrix $\begin{bmatrix} \tilde{A}_{22} \\ \tilde{A}_{32} \end{bmatrix}$ to get

$$\begin{bmatrix} \tilde{A}_{22} \\ \tilde{A}_{32} \end{bmatrix} = \begin{bmatrix} L_{22} \\ L_{32} \end{bmatrix} \cdot U_{22} = \begin{bmatrix} L_{22}U_{22} \\ L_{32}U_{22} \end{bmatrix}.$$

This lets us write

$$\begin{bmatrix} \tilde{A}_{22} & \tilde{A}_{23} \\ \tilde{A}_{32} & \tilde{A}_{33} \end{bmatrix} = \begin{bmatrix} L_{22}U_{22} & \tilde{A}_{23} \\ L_{32}U_{22} & \tilde{A}_{33} \end{bmatrix}$$

$$= \begin{bmatrix} L_{22} & 0 \\ L_{32} & I \end{bmatrix} \cdot \begin{bmatrix} U_{22} & L_{22}^{-1}\tilde{A}_{23} \\ 0 & \tilde{A}_{33} - L_{32} \cdot (L_{22}^{-1}\tilde{A}_{23}) \end{bmatrix}$$
$$\equiv \begin{bmatrix} L_{22} & 0 \\ L_{32} & I \end{bmatrix} \cdot \begin{bmatrix} U_{22} & U_{23} \\ 0 & \tilde{A}_{33} - L_{32} \cdot U_{23} \end{bmatrix}$$
$$\equiv \begin{bmatrix} L_{22} & 0 \\ L_{32} & I \end{bmatrix} \cdot \begin{bmatrix} U_{22} & U_{23} \\ 0 & \tilde{\tilde{A}}_{33} \end{bmatrix}.$$

Altogether, we get an updated factorization with b more columns of L and rows of U completed:

$$\begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{bmatrix} = \begin{bmatrix} L_{11} & 0 & 0 \\ L_{21} & L_{22} & 0 \\ L_{31} & L_{23} & I \end{bmatrix} \cdot \begin{bmatrix} U_{11} & U_{21} & U_{31} \\ 0 & U_{22} & U_{23} \\ 0 & 0 & \tilde{\tilde{A}}_{33} \end{bmatrix}.$$

This defines an algorithm with the following three steps, which are illustrated on the right of Figure 2.6:

(1) Use Algorithm 2.9 to factorize $\begin{bmatrix} \tilde{A}_{22} \\ \tilde{A}_{32} \end{bmatrix} = \begin{bmatrix} L_{22} \\ L_{32} \end{bmatrix} \cdot U_{22}$.

(2) Form $U_{23} = L_{22}^{-1}\tilde{A}_{23}$. This means solving a triangular linear system with many right-hand sides ($\tilde{A}_{23}$), a single Level 3 BLAS operation.

(3) Form $\tilde{\tilde{A}}_{33} = \tilde{A}_{33} - L_{32} \cdot U_{23}$, a matrix-matrix multiplication.

More formally, we have the following algorithm.

ALGORITHM 2.10. *Level 3 BLAS implementation of LU factorization without pivoting for an n-by-n matrix A. Overwrite L and U on A. The lines of the algorithm are numbered as above and to correspond to the right part of Figure 2.6.*

for $i = 1$ *to* $n-1$ *step* b

(1) *Use Algorithm 2.9 to factorize* $A(i:n, i:i+b-1) = \begin{bmatrix} L_{22} \\ L_{32} \end{bmatrix} U_{22}$

(2) $A(i:i+b-1, i+b:n) = L_{22}^{-1} \cdot A(i:i+b-1, i+b:n)$
/ form* U_{23} **/*

(3) $A(i+b:n, i+b:n) = A(i+b:n, i+b:n)$
$-A(i+b:n, i:i+b-1) \cdot A(i:i+b-1, i+b:n)$
/ form* $\tilde{\tilde{A}}_{33}$ **/*

end for

We still need to choose the block size b in order to maximize the speed of the algorithm. On the one hand, we would like to make b large because we have seen that speed increases when multiplying larger matrices. On the other hand, we can verify that the number of floating point operations performed

by the slower Level 2 and Level 1 BLAS in line (1) of the algorithm is about $n^2b/2$ for small b, which grows as b grows, so we do not want to pick b too large. The optimal value of b is machine dependent and can be tuned for each machine. Values of $b = 32$ or $b = 64$ are commonly used.

To see detailed implementations of Algorithms 2.9 and 2.10, see subroutines `sgetf2` and `sgetrf`, respectively, in LAPACK (NETLIB/lapack). For more information on block algorithms, including detailed performance number on a variety of machines, see also [10] or the course notes at PARALLEL_HOMEPAGE.

2.6.4. More About Parallelism and Other Performance Issues

In this section we briefly survey other issues involved in implementing Gaussian elimination (and other linear algebra routines) as efficiently as possible.

A *parallel computer* contains $p > 1$ processors capable of simultaneously working on the same problem. One may hope to solve any given problem p times faster on such a machine than on a conventional uniprocessor. But such "perfect efficiency" is rarely achieved, even if there are always at least p independent tasks available to do, because of the overhead of coordinating p processors and the cost of sending data from the processor that may store it to the processor that needs it. This last problem is another example of a *memory hierarchy*: from the point of view of processor i, its own memory is fast, but getting data from the memory owned by processor j is slower, sometimes thousands of times slower.

Gaussian elimination offers many opportunities for parallelism, since each entry of the trailing submatrix may be updated independently and in parallel at each step. But some care is needed to be as efficient as possible. Two standard pieces of software are available. The LAPACK routine `sgetrf` described in the last section [10] runs on *shared-memory parallel machines*, provided that one has available implementations of the BLAS that run in parallel. A related library called ScaLAPACK, for *Scalable LAPACK* [34, 53], is designed for *distributed-memory parallel machines*, i.e., those that require special operations to move data between different processors. All software is available on NETLIB in the LAPACK and ScaLAPACK subdirectories. ScaLAPACK is described in more detail in the notes at PARALLEL_HOMEPAGE. Extensive performance data for linear equation solvers are available as the LINPACK Benchmark [85], with an up-to-date version available at NETLIB/benchmark/performance.ps, or in the Performance Database Server.[14] As of May 1997, the fastest that any linear system had been solved using Gaussian elimination was one with $n = 215000$ on an Intel ASCI Option Red with $p = 7264$ processors; the problem ran at just over 1068 Gflops (gigaflops), out of a maximum 1453 Gflops.

[14] http://performance.netlib.org/performance/html/PDStop.html

There are some matrices too large to fit in the main memory of any available machine. These matrices are stored on disk and must be read into main memory piece by piece in order to perform Gaussian elimination. The organization of such routines is largely similar to the technique described above, and they are included in ScaLAPACK.

Finally, one might hope that compilers would become sufficiently clever to take the simplest implementation of Gaussian elimination using three nested loops and automatically "optimize" the code to look like the blocked algorithm discussed in the last subsection. While there is much current research on this topic (see the bibliography in the recent compiler textbook [264]), there is still no reliably fast alternative to optimized libraries such as LAPACK and ScaLAPACK.

2.7. Special Linear Systems

As mentioned in section 1.2, it is important to exploit any special structure of the matrix to increase speed of solution and decrease storage. In practice, of course, the cost of the extra programming effort required to exploit this structure must be taken into account. For example, if our only goal is to minimize the time to get the desired solution, and it takes an extra week of programming effort to decrease the solution time from 10 seconds to 1 second, it is worth doing only if we are going to use the routine more than (1 week * 7 days/week * 24 hours/day * 3600 seconds/hour) / (10 seconds − 1 second) = 67200 times. Fortunately, there are some special structures that turn up frequently enough that standard solutions exist, and we should certainly use them. The ones we consider here are

1. s.p.d. matrices,
2. symmetric indefinite matrices,
3. band matrices,
4. general sparse matrices,
5. dense matrices depending on fewer than n^2 independent parameters.

We will consider only real matrices; extensions to complex matrices are straightforward.

2.7.1. Real Symmetric Positive Definite Matrices

Recall that a real matrix A is s.p.d. if and only if $A = A^T$ and $x^T Ax > 0$ for all $x \neq 0$. In this section we will show how to solve $Ax = b$ in half the time and half the space of Gaussian elimination when A is s.p.d.

PROPOSITION 2.2. 1. *If X is nonsingular, then A is s.p.d. if and only if $X^T AX$ is s.p.d.*

2. *If A is s.p.d. and H is any principal submatrix of A ($H = A(j:k, j:k)$ for some $j \leq k$), then H is s.p.d.*
3. *A is s.p.d. if and only if $A = A^T$ and all its eigenvalues are positive.*
4. *If A is s.p.d., then all $a_{ii} > 0$, and $\max_{ij} |a_{ij}| = \max_i a_{ii} > 0$.*
5. *A is s.p.d. if and only if there is a unique lower triangular nonsingular matrix L, with positive diagonal entries, such that $A = LL^T$. $A = LL^T$ is called the* Cholesky factorization *of A, and L is called the* Cholesky factor *of A.*

Proof.

1. X nonsingular implies $Xx \neq 0$ for all $x \neq 0$, so $x^T X^T AXx > 0$ for all $x \neq 0$. So A s.p.d. implies $X^T AX$ is s.p.d. Use X^{-1} to deduce the other implication.
2. Suppose first that $H = A(1:m, 1:m)$. Then given any m-vector y, the n-vector $x = [y^T, 0]^T$ satisfies $y^T Hy = x^T Ax$. So if $x^T Ax > 0$ for all nonzero x, then $y^T Hy > 0$ for all nonzero y, and so H is s.p.d. If H does not lie in the upper left corner of A, let P be a permutation so that H does lie in the upper left corner of $P^T AP$ and apply Part 1.
3. Let X be the real, orthogonal eigenvector matrix of A so that $X^T AX = \Lambda$ is the diagonal matrix of real eigenvalues λ_i. Since $x^T \Lambda x = \sum_i \lambda_i x_i^2$, Λ is s.p.d if and only if each $\lambda_i > 0$. Now apply Part 1.
4. Let e_i be the ith column of the identity matrix. Then $e_i^T Ae_i = a_{ii} > 0$ for all i. If $|a_{kl}| = \max_{ij} |a_{ij}|$ but $k \neq l$, choose $x = e_k - \text{sign}(a_{kl})e_l$. Then $x^T Ax = a_{kk} + a_{ll} - 2|a_{kl}| \leq 0$, contradicting positive-definiteness.
5. Suppose $A = LL^T$ with L nonsingular. Then $x^T Ax = (x^T L)(L^T x) = \|L^T x\|_2^2 > 0$ for all $x \neq 0$, so A is s.p.d. If A is s.p.d., we show that L exists by induction on the dimension n. If we choose each $l_{ii} > 0$, our construction will determine L uniquely. If $n = 1$, choose $l_{11} = \sqrt{a_{11}}$, which exists since $a_{11} > 0$. As with Gaussian elimination, it suffices to understand the block 2-by-2 case. Write

$$
\begin{aligned}
A &= \begin{bmatrix} a_{11} & A_{12} \\ A_{12}^T & A_{22} \end{bmatrix} \\
&= \begin{bmatrix} \sqrt{a_{11}} & 0 \\ \frac{A_{12}^T}{\sqrt{a_{11}}} & I \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & \tilde{A}_{22} \end{bmatrix} \begin{bmatrix} \sqrt{a_{11}} & \frac{A_{12}}{\sqrt{a_{11}}} \\ 0 & I \end{bmatrix}
\end{aligned}
$$