Chapter 3
Matrix Computational Concerns

We give some basics on the analysis and design of numerical algorithms for linear algebra computations in floating point arithmetic. Basic notions about computing in floating point arithmetic are given in §3.1. In §3.5, we give examples showing the influence of the ideas in §3.1 on the design of numerical algorithms. A more thorough discussion of this general topic is given in the references at the end of the chapter.

3.1. Floating Point Computation

3.1.1. A simple model of floating point arithmetic

The most common method of representing the real number system on digital computers is floating point arithmetic. The following is a simplified model.

A floating point number $x$ has the form

$$ x = \pm d \cdot 2^e $$

where $\beta$ is an integer base, $e$ is an integer exponent, and $d \in [0,1)$ is the mantissa. For our purposes, all arithmetic will be in base 2 since that is standard on almost all general purpose computers. Some special purpose devices such as pocket calculators, use base 10.

Any real number can be represented in the form (3.1). In floating point arithmetic, there are limitations on $d$ and $e$. The exponent $e$ satisfies the bound

$$ e_{\min} \leq e \leq e_{\max}, $$

where

$$ e_{\min} < 0 < e_{\max}. $$

The mantissa $d$ has the form

$$ d = 0.d_1d_2 \ldots d_t = d_1 2^{-1} + d_2 2^{-2} + \cdots + d_t 2^{-t}, $$

where $d_i \in \{0, 1\}$ and $t$ is the number of digits of precision in the floating point system. A floating point number with $d_1 \neq 0$ is said to be normalized and is
said to be unnormalized if \( d_1 = 0 \). In our model, a floating point number is normalized if and only if \( |x| \geq 2^{e_{\min} - 1} \). Thus only the numbers at the bottom of the exponent range are unnormalized.

Let \( \text{fl}(\cdot) \) be a function which denotes the floating point computation of the contents and let \( x^* = \text{fl}(x) \) be the floating point number rounded from \( x \). The machine unit \( \varepsilon_M \) is defined by

\[
\varepsilon_M = \max_{[\log_2|x|] \in [e_{\min}, e_{\max}]} \frac{|x - \text{fl}(x)|}{|x|}.
\]

(3.4)

The rule for obtaining \( \text{fl}(x) \) from \( x \) varies. The most common rule, standard rounding chooses the floating point number nearest to \( x \) along with some specification for breaking ties. For that rule, a simple computation shows that \( \varepsilon_M = 2^{-t} \).

The floating point versions of the operations addition, subtraction, multiplication, and division satisfy

\[
\text{fl}(x \text{ op } y) = (x \text{ op } y)(1 + \xi), \quad |\xi| \leq \varepsilon_M,
\]

(3.5)

where \( \text{op} \in \{+,-,\times,\div\} \) unless the result is outside the exponent range.

For a few machines, the addition and subtraction operations do not obey (3.5), but instead obey the somewhat weaker rounding rule

\[
\text{fl}(x \pm y) = x(1 + \xi_1) \pm y(1 + \xi_2), \quad |\xi_i| \leq \varepsilon_M, \quad i = 1, 2.
\]

(3.6)

At this writing, CRAY was the only major computer manufacturer who arithmetic obeys the rule (3.6), but not the rule (3.5). All analysis in this book will assume the rounding rule (3.5). For further discussion see [?, 2002, §2.4].

The exponent range is the another limitation of floating point computation. An overflow is a result whose magnitude is greater than the largest computer number, denoted by \( \Omega \). In terms of \( t \), and \( e_{\max} \),

\[
\Omega = (1 - 2^{-t})2^{e_{\max}}.
\]

In IEEE arithmetic (to be discussed in §3.1.2) an overflow results in one of the special numbers \( \text{Inf} \) or \( -\text{Inf} \).

An underflow is a result whose magnitude is smaller than the smallest nonzero computer number, denoted \( \omega \).

We now give two methods for determining \( \omega \) as classified by Demmel [10, 1984]. One of them is called “store zero” (SZ). For SZ, the smallest computer number is

\[
\omega = \beta^{-1}\beta^{e_{\min}} = \beta^{1-t}\omega_{SZ}.
\]

(3.7)

The smallest “useful” nonzero computer number is

\[
\omega_{useful} = \beta^{-1}\beta^{e_{\min}}.
\]

(3.8)
3.1 Floating Point Computation

This is actually the smallest normalized floating point number. Before the IEEE floating point standard discussed in the next section, most computers made this the smallest nonzero floating point number. These computers required all numbers such that \( |\log_2| \leq e_{\text{min}} \) be set to zero.

The advantage of the philosophy embodied in (3.7) called gradual underflow, is a slightly expanded number range. A more important advantage is that

\[
\text{fl}(x - y) = 0 \quad \text{if and only if } x = y.
\]

Demmel [10, 1984] gives examples where each of the two philosophies and makes a case for gradual underflow.

If underflow is included in error analyses, (3.5) becomes

\[
\text{fl}(x \text{ op } y) = (x \text{ op } y)(1 + \xi) + \eta
\]

where

\[
|\xi| \leq \varepsilon_M, \quad |\eta| \leq \omega, \quad \eta \cdot \xi = 0.
\]

Note that at least one of the values \( \xi \) and \( \eta \) is zero.

For underflows, the programmer has a choice as to whether they are set to zero or to generate an error flag. In some expert codes, underflows generate flags in some parts of the codes, but be set to zero in others.

In subsequent error analyses, unless otherwise stated, we ignore the effect of underflows or overflows.

General purpose computers implement at least two types of floating point arithmetic, single precision and double precision. Single precision is specified with precision \( t_s \), minimum exponent \( e_s^{\text{min}} \), and maximum exponent \( e_s^{\text{max}} \) and double precision is specified by \( t_d, e_d^{\text{min}} \) and \( e_d^{\text{max}} \) respectively. Presumably the base \( \beta \) is the same for both precisions. Double precision usually has a much larger exponent range than single precision, thus we expect \( e_d^{\text{min}} \ll e_s^{\text{min}} \) and \( e_d^{\text{max}} \gg e_s^{\text{max}} \).

A third precision, extended precision, is implemented on many computers. Extended precision would again have an expanded exponent range and its arithmetic would have a machine unit \( \varepsilon_M^{\text{ext}} \) satisfying

\[
\varepsilon_M^{\text{ext}} \leq c_2(\varepsilon_M)^4.
\]

for some reasonable sized constant \( c_2 \).

If extended precision is unavailable on a given computer, it can be simulated provided the computer arithmetic uses guard digits, which are extra digits used to determine a correct round. (Again, CRAY machines do not.) A set of routines for simulating extended precision arithmetic operations in double precision (or double precision operations in single precision) for base \( \beta = 2 \) is given by Dekker [9, 1971] and Kahan [22, 1972]. They are short and easy to program, but require the value of \( t \).
3.1.2. The IEEE Floating Point Standard

Eccentricities in the implementation of floating point arithmetic have been a significant barrier to the development of robust, portable numerical software. The biggest problems have been subtle differences in rounding from the idealized round in the last section and the handling of numbers at the extreme of the number range.

In the early 1970’s, a hodge-podge of rounding rules were implemented (see Sterbenz[27, 1974], Cody[8, 1973], and Barlow[2, 1981]). A user often had to have a detailed knowledge of the arithmetic in a computer to produce reliable numerical software. Moreover, drastic changes might be necessary for a robust program on one computer to become a robust program on another. A strong, emphatic argument for a floating point standard is given by Kahan [25, 1981].

In 1977, IEEE task P754 was established to produce a standard for floating point arithmetic. It produced the first such standard in December 1982 which was approved two years later as ANSI/IEEE Std 754–1985, the IEEE standard for binary floating point arithmetic. A second task, P854, was organized in January 1981 to generalize and extend the binary standard. The effort ended in March 1987 with the formal approval by the IEEE Standard Board of IEEE Std 854–1987, A radix independent standard for floating point arithmetic.

The binary standard, ANSI/IEEE Std 754–1985, is very specific. It specifies length, precision, and exponent range. Thus all computers implementing the binary standard will have identical exponent range and precision for the implementations of single, double, and (where supported) extended precision.

Most computers now manufactured including mainframe computers, workstations, and personal computers implement ANSI/IEEE 754–1985. The principal exceptions are the CRAY X-MP,Y-MP,C90, and J90 (see Kahan [24, 1995]) which are fading from the scene.

As is also pointed out in [24, 1995], a program will not necessarily obtain exactly the same answers on two different computers that implement the binary standard. There are subtle variations among implementations, but, minimally, all implementations should fit the specifications given below.

The radix independent standard, IEEE Std 854–1987, is less specific and relies on a parameterization, but, except for pocket calculators, the binary standard, 754–1985, is the one of interest.

The following specifications are the most used.

- All underflows are implemented using gradual underflow as discussed in §3.1.1. Thus \( f(t(x - y)) = 0 \) if and only if \( x = y \).

- The parameters specifying single precision are \( t = 24, e_{\text{max}} = 128 \), and \( e_{\text{min}} = -126 \).
3.1 Floating Point Computation

- Double precision is required. The specifications for double precision are \( t_d = 53, e_{max}^d = 1024, e_{min}^d = -1022 \).

If \( \varepsilon_M^s \) and \( \varepsilon_M^d \) are the machine units for single and double precision, respectively, then IEEE arithmetic assures us that

\[
\varepsilon_M^d < (\varepsilon_M^s)^2
\]  

(3.11)
as mixed precision error analysis would assume.

As well as supporting the range of floating point numbers specified in (3.1)–(3.2), each precision must also represent

1. Two infinities \(+\infty\) (or \( \text{Inf} \)) and \(-\infty\) (or \( \text{Inf} \)).

2. A signaling \( \text{NaN} \) (not a number) and a quiet \( \text{NaN} \).

3. Two zeros \(+0\) and \(-0\).

The first two specifications are designed to make IEEE arithmetic a closed system. Every operation with standard floating point numbers, infinities, or \( \text{NaNs} \) results in standard floating point numbers, infinities, or \( \text{NaNs} \). The need for \(+0\) and \(-0\) is for the proper implementation of branch cuts as discussed by Kahan [23, 1996].

Arithmetic on \( \infty \) is always exact and, in itself, signals no exceptions, unless \( \infty \) is created by overflow or division by zero or used in the following invalid operations

1. \((+\infty) + (-\infty)\).

2. \(0 \times \infty\).

3. \(\infty/\infty\).

4. Remainder, \(x \ REM y\) where \(x = \pm\infty\) and \(y = 0\).

The special value \( \text{NaN} \) literally means “not a number.” A signaling \( \text{NaN} \) is a reserved operand that always signals an invalid exception. All four of the invalid operations above yield a signaling \( \text{NaN} \). Signaling \( \text{NaNs} \) are also created by square roots of negative numbers (except for \(-0\)) and the operation \(\pm0/\pm0\). A quiet \( \text{NaN} \) is used for diagnostic information about invalid or unavailable information. Quiet \( \text{NaNs} \) would be propagated throughout the computation. A \( \text{NaN} \) is not required to have an algebraic sign.

IEEE arithmetic supports four rounding modes

1. **Round to nearest.** This is standard rounding. All overflows are carried to \( \infty \) or \(-\infty \).
2. **Round toward zero.** This is standard chopping. All overflows are carried to $\Omega$ or $-\Omega$.

3. **Round toward $-\infty$.** All positive overflows are carried to $\Omega$ and negative overflows are carried to $-\infty$.

4. **Round toward $\infty$.** All positive overflows are carried to $\infty$ and negative overflows are carried to $-\Omega$.

There are other specifications to IEEE arithmetic that include input and output and number conversions. For complete details, refer to the original papers [21, 1981], [19, 1985], and [20, 1987], an article by Goldberg [?, 1997], and a book by Overton [?, 2001].

### 3.1.3. Error Analysis of Simple Computations

As a matter of course, we will not give error analyses of algorithms. Such analyses, somewhat like program correctness proofs, tend to be long and technical. Below we show some simple examples of how error analyses go. More such analyses are given in detail in Wilkinson [32, 1965], Demmel [?, 1997], Higham [?, 2002]. The Higham book has a wonderful reference list.

Consider the floating point product

$$p_n = x_1 x_2 \ldots x_n,$$

where $x_i, i = 1, 2, \ldots, n$ are floating point numbers. We use the most obvious algorithm to compute $p_n$, namely

$$p_1 = x_1,$$

$$p_{k+1} = p_k x_{k+1}, \quad k = 1, 2, \ldots, n-1.$$

The floating point products $\tilde{p}_k = fl(p_k)$ are given by

$$\tilde{p}_1 = p_1 = x_1,$$

$$\tilde{p}_{k+1} = fl(\tilde{p}_k x_{k+1}) = \tilde{p}_k x_{k+1}(1 + \xi_k), \quad |\xi_k| \leq \varepsilon_M,$$

$$k = 1, 2, \ldots, n-1.$$

Thus, by an induction argument

$$\tilde{p}_n = p_n (1 + \eta)$$

where

$$1 + \eta = \prod_{k=1}^{n-1} (1 + \eta_k).$$
3.1 Floating Point Computation

It follows that

\[(1 - \varepsilon_M)^{n-1} - 1 \leq \eta \leq (1 + \varepsilon_M)^{n-1} - 1.\]  \hspace{1cm} (3.15)

Since \(\varepsilon_M\) is a small number, we use the simplification

\[|(1 \pm \varepsilon_M)^{k-1} - 1| = (k - 1)\varepsilon_M + O(\varepsilon_M^2).\]

Thus \(\eta\) in (3.15) satisfies

\[|\eta| \leq (n - 1)\varepsilon_M + O(\varepsilon_M^2).\]  \hspace{1cm} (3.16)

The same bound can be given if divisions are substituted for multiplications. Thus a long sequence of multiplications or divisions always satisfies a relative error bound.

A floating point sum is a different matter. Let

\[s_n = \sum_{i=1}^{n} x_i.\]  \hspace{1cm} (3.17)

The usual algorithm to compute \(s_n\) is the recurrence

\[s_1 = x_1,\]  \hspace{1cm} (3.18)

\[s_{k+1} = s_k + x_{k+1}, \quad k = 1, 2, \ldots, n - 1.\]  \hspace{1cm} (3.19)

The floating point version of (3.18)–(3.19) yields the sequence \(\tilde{s}_k = \text{fl}(s_k)\) given by

\[\tilde{s}_1 = s_1 = x_1,\]  \hspace{1cm} (3.20)

\[\tilde{s}_{k+1} = \text{fl}(\tilde{s}_k + x_{k+1}) = (\tilde{s}_k + x_{k+1})(1 + \xi_k), \quad k = 1, 2, \ldots, n - 1,\]  \hspace{1cm} (3.21)

where

\[|\xi_k| \leq \varepsilon_M, \quad k = 1, 2, \ldots, n - 1.\]  \hspace{1cm} (3.22)

Thus

\[\tilde{s}_n = \sum_{i=1}^{n} x_i(1 + \eta_i)\]  \hspace{1cm} (3.23)

where

\[1 + \eta_i = \prod_{k=i-1}^{n-1} (1 + \xi_k), \quad i = 2, \ldots, n,\]

\[\eta_1 = \eta_2,\]

resulting in the bounds

\[|\eta_i| \leq (n - i + 1)\varepsilon_M + O(\varepsilon_M^2), \quad i = 2, \ldots, n,\]  \hspace{1cm} (3.24)
and
\[ |\eta_1| \leq (n - 1) \varepsilon_M + O(\varepsilon_M^2). \] (3.25)

Unlike floating point products, there is no reasonable bound on
\[ \frac{|\tilde{s}_n - s_n|}{|s_n|}. \]
A reasonable bound on \(|\tilde{s}_n - s_n|\) is
\[ |\tilde{s}_n - s_n| \leq \max_{1 \leq i \leq n} |\eta_i| \sum_{i=1}^{n} |x_i|, \]
\[ \leq (n - 1) \varepsilon_M \|x\|_1 + O(\varepsilon_M^2) \]
where \(x = (x_1, x_2, \ldots, x_n)^T\). If \(|s_n| \ll \|x\|_1\), little relative accuracy can be expected of this procedure.

Differences in how the sums are ordered can obtain a smaller error bound. For instance, if \(x_1 \geq x_2 \geq \ldots \geq x_n > 0\)
from (3.23)-(3.24), we can easily see that it is better to compute the backward recurrence
\[ t_1 = x_n, \]
\[ t_{k+1} = t_k + x_{n-k}, \quad k = 1, 2, \ldots, n-1, \]
and then let \(s_n = t_n\).

Another algorithm to compute \(s_n\) could obtain a slightly different error bound. The following technique is common for computation on distributed computers [30, 1995, pp. 26-33]. Let
\[ s_{ij} = \sum_{k=i}^{j} x_k, \quad 1 \leq i < j \leq n. \]
We can then compute these sum according to
\[ s_{1,m+\ell} = s_{1,m} + s_{m+1,m+\ell}. \]
Since \(s_n = s_{1,n}\) we can compute \(s_n\) from the following recursive procedure.

**Function 3.1 (Recursive Pairwise Summation)**
3.1 Floating Point Computation

\[ s_n \leftarrow \text{sum}(x, 1, n) \]
function \( s = \text{sum}(x, i, j) \)
if \( i = j \) then
  \( \text{sum} = x(i) \)
else
  \( k = [(i + j)/2] \)
  \( \text{sum} = \text{sum}(x, i, k) + \text{sum}(x, k + 1, j) \)
end if
end sum

The value \( s_n \) is computed in \([\log_2 n]\) levels of recursion. The \([\log_2 n]\) is also important in bounding the backward error. This algorithm can be shown to produce a sum \( \hat{s}_n \) satisfying (3.23) where

\[ |\eta| \leq [\log_2 n] \varepsilon_M + O(\varepsilon_M^2). \]  

(3.26)

If \( n \) is very large and all of the \( x_i \) are of about the same magnitude, the recursive procedure will yield an better forward error bound than (3.18)–(3.19). Other subtle differences in summing strategies are discussed in [? , 2003, Chapter 4].

The important point of these error bounds is that the computed sum \( \hat{s}_n \) is the exact sum

\[ \hat{s}_n = \sum_{i=1}^{n} \hat{x}_i \]

where \( \hat{x}_1, \hat{x}_2, \ldots, \hat{x}_n \) are close to \( x_1, x_2, \ldots, x_n \). Thus both of the algorithms described here are backward stable.

A very similar argument will yield an error bound on the Euclidean dot product \( x^T y \) that is given by

\[ |f(x^T y) - x^T y| \leq g(n) \varepsilon_M |x| |y| + O(\varepsilon_M^2) \]  

(3.27)

where \( g(n) = n \) is standard summation is used and \( g(n) = [\log_2 n] + 1 \) if pairwise summation is used. If \( |x| |y| \gg |x^T y| \), then relative accuracy cannot be guaranteed for the dot product.

Likewise, if we compute \( C = AB \), where \( A \in \mathbb{R}^{m \times n} \), \( B \in \mathbb{R}^{n \times p} \) and \( C \in \mathbb{R}^{m \times p} \) we have the error bound

\[ |f(AB) - AB| \leq g(n) \varepsilon_M |A| |B| + O(\varepsilon_M^2). \]  

(3.28)

If, for a particular \((i, j)\), \( |c_{ij}| \ll |e_i^T A| |B e_j| \) then \( c_{ij} \), then relative accuracy will not be guaranteed for \( c_{ij} \).

One method for improving the bounds (3.27) or (3.28) is double precision accumulation of inner products. We assume that our double precision satisfies (3.11). Double precision accumulation is the process of taking two single
precision vectors, \( \mathbf{x} \) and \( \mathbf{y} \), copying them into double precision vectors, computing the inner product \( \mathbf{x}^T \mathbf{y} \) in double precision, and rounding the result into single precision. We use the notation \( fl_2(\cdot) \) to denote a double precision accumulated computation.

We have that

\[
fl_2(\mathbf{x}^T \mathbf{y}) = [\mathbf{x}^T \mathbf{y} + \eta \mathbf{x}^T \mathbf{y}] (1 + \eta_2)
\]

where

\[
|\eta| \leq n \varepsilon_{M^n}, \quad |\eta_2| \leq \varepsilon_{M^2}.
\]

If

\[
\frac{|\mathbf{x}^T \mathbf{y}|}{|\mathbf{x}|^T |\mathbf{y}|} \geq \varepsilon_{M^n},
\]

then the dot product \( \mathbf{x}^T \mathbf{y} \) is computed to near relative accuracy in single precision.

If we compute two-norm of a vector \( \mathbf{x} \) using the formula

\[
\|\mathbf{x}\|_2 = (\mathbf{x}^T \mathbf{x})^{1/2},
\]

then the bound (3.27), the formula \( \sqrt{1 + \eta} = 1 + 0.5\eta + O(\eta^2) \), and a single round for the square root yield the bound

\[
|\text{fl}(\|\mathbf{x}\|_2) - \|\mathbf{x}\|_2| \leq (0.5g(n) + 1)\varepsilon_M \|\mathbf{x}\|_2 + O(\varepsilon_{M}^2)
\] (3.29)

where \( g(n) \) is the same function as in (3.27). Thus we expect that the two-norm of a vector will be computed to near relative accuracy. In §3.5.2, we show that the computational issues concerning the two-norm do not end with equation (3.29).

### 3.1.4. Stability and Computation

The limitations of floating point arithmetic impose limitations upon our ability to compute a real vector valued function \( \mathbf{g} : \mathbb{R}^n \to \mathbb{R}^m \). For a particular, \( \mathbf{x} \in \mathbb{R}^n \), there are four barriers between the value obtained for \( \mathbf{g}(\mathbf{x}) \) from a computer algorithm and the mathematical value of \( \mathbf{g}(\mathbf{x}) \).

- **Number range of the computer.** If \( x_i \), the \( i \)th component of \( \mathbf{x} \), is larger in magnitude than \( \Omega \), the largest computer number, then we cannot represent it or must use some trick to represent it. If \( |x_i| < \omega \), then we may either represent it as zero or use some trick to represent it.

- **Input or representation errors.** The components of \( \mathbf{x} \), \( x_1, \ldots, x_n \), cannot be represented exactly. They must be rounded to form a nearby machine representable vector, \( \tilde{\mathbf{x}} = (\tilde{x}_1, \ldots, \tilde{x}_n)^T \). Thus a computer algorithm to compute \( \mathbf{g}(\mathbf{x}) \), in fact, tries to compute \( \mathbf{g}(\tilde{\mathbf{x}}) \).
3.1 Floating Point Computation

- **Approximation errors.** Computers have only the operations addition, subtraction, multiplication, division, and square root. This means that we are sometimes computing an approximate function \( \tilde{g}(\tilde{x}) \). We may also choose to compute an approximate function if the computation of \( g(\tilde{x}) \) is too expensive.

- **Rounding errors.** The operations addition, subtraction, multiplication, division, and square root are not exact. Their results are rounded to machine representable numbers. Thus even \( \tilde{g} \), the approximate function is approximated by a machine version \( \tilde{g} \).

We refer to \( \tilde{g}(\tilde{x}) \) as the value produced by a computer algorithm to compute \( g(x) \).

If component of the vector \( x \) are out of the computer range, they can be rescaled. Input or representation errors can be resolved by the theory of conditioning [16, 1989], [28, 1973], [29, 1990], the same theory we use for rounding errors.

We discuss two methods for analyzing rounding errors, **forward error analysis** and **backward error analysis**.

In forward error analysis, we obtain a bound on the computed function \( \tilde{g} \) of the form

\[
||\tilde{g}(\tilde{x}) - g(x)|| \leq \delta(x)
\]

where \( \delta(x) \) is some "small" function of \( x \) and ||·|| is some norm. An algorithm satisfying (3.30) is said to be **strongly forward stable**. Bounds such as (3.30) are very difficult to obtain. The algorithm for the floating point product (3.12) given by (3.13)–(3.14) is strongly forward stable.

In backward error analysis, we find a vector \( \tilde{x} \) such that

\[
\tilde{g}(\tilde{x}) = g(\tilde{x}).
\]

\[
||\tilde{x} - x|| \leq \eta(x)
\]

for some small function \( \eta(\cdot) \). Since representation errors would be expected to cause an error such as (3.31)–(3.32), we may reasonably say that this sort of bound is as good as we can expect from any algorithm. Our analysis of floating point sums obtained a backward error bound.

The bound (3.30) assures that the computed function \( \tilde{g} \) is close to \( g \), the bound (3.31)–(3.32) does not. We need the additional assumption for some constant \( \kappa_g \) and for all \( \tilde{x} \) satisfying (3.32)

\[
||\tilde{g}(\tilde{x}) - g(x)|| \leq \kappa_g ||\tilde{x} - x||.
\]

The constant \( \kappa_g \) in (3.33) is important. If \( \kappa_g \) is not too large, we consider \( g \) to be well-conditioned around \( x \). If \( \kappa_g \) is large, then \( g \) is considered ill-conditioned. If no bound of the form (3.33) exists, the computation of \( g \) is ill-posed.
Backward error analysis separates the analysis of the solution of $g$ into two separate problems. The first is to show that a particular algorithm satisfies (3.31)-(3.32) and the second is to show that a bound of the form (3.33) exists. If a bound of the form (3.31)-(3.32) is obtained for an algorithm for computing $g$ we say that the algorithm is *backward stable*. The algorithms given in §3.1.3 for sums and inner products are all backward stable, but are not strongly forward stable.

For floating point sums the function $g$ is given by

$$g(x) = \sum_{i=1}^{n} x_i$$

and we have that

$$|g(\tilde{x}) - g(x)| \leq \kappa_g \|\tilde{x} - x\|$$

where $\kappa_g$ is dependent upon the choice of norm $\| \cdot \|$. For the three most commonly used vector norms, the values of $\kappa_g$ are

$$\kappa_g = \begin{cases} 1 & \text{if } \| \cdot \| = \| \cdot \|_1 \\ \sqrt{n} & \text{if } \| \cdot \| = \| \cdot \|_2 \\ n & \text{if } \| \cdot \| = \| \cdot \|_\infty \end{cases} .$$

Subtle variations of backward stability are used throughout the numerical linear algebra literature. Below are two such variations.

An algorithm $\tilde{g}$ to compute $g$ is considered *numerically stable* if

$$\|\tilde{g}(x) - g(\tilde{x})\| \leq \delta(x)$$

for some $\tilde{x}$ satisfying (3.32) even if there is no $\tilde{x}$ such that $g(\tilde{x}) = \tilde{g}(x)$. This is a mixture of backward and forward stability. In practice, there is little difference between an algorithm satisfying (3.34) and a backward stable algorithm.

We sometimes have a bound of the form

$$\|\tilde{g}(x) - g(x)\| \leq \kappa_g \eta(x)$$

where $\eta(x)$ is as in (3.32), but $\tilde{g}(x)$ does not satisfy (3.34) for any small function $\delta(x)$. This is called *weak forward stability* and means that the algorithm obtains a forward error bound that would be expected of a backward stable algorithm.

An algorithm for which the best error bound is significantly weaker than (3.35) is called *unstable*. The use of such algorithms should be avoided.
3.2. Solution of Triangular Linear Systems

The solution of triangular linear systems is a common operation in matrix computations. We review the basic algorithms for them here.

A square matrix \( L = (\ell_{ij}) \in \mathbb{R}^{n \times n} \) is lower triangular if

\[
\ell_{ij} = 0, \quad i < j.
\]

(3.36)

It may be written

\[
L = \begin{pmatrix}
\ell_{11} & 0 & \cdots & 0 \\
\ell_{21} & \ell_{22} & \cdots & 0 \\
\vdots & \vdots & \ddots & \vdots \\
\ell_{n1} & \ell_{n2} & \cdots & \ell_{nn}
\end{pmatrix}.
\]

By induction on the relation

\[
\det(L) = \ell_{11} \det(L_{22})
\]

we have that

\[
\det(L) = \ell_{11} \ell_{22} \cdots \ell_{nn}.
\]

Thus \( L \) is nonsingular if and only if \( \ell_{ii} \neq 0 \) for all \( i \).

We now show how to solve

\[
Lx = b.
\]

(3.37)

Consider the partitioning

\[
L = \frac{1}{n-1} \begin{pmatrix}
\ell_{11} & 0 \\
\ell_{21} & L_{22}
\end{pmatrix}, \quad x = \frac{1}{n-1} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}, \quad b = \frac{1}{n-1} \begin{pmatrix} b_1 \\ b_2 \end{pmatrix}.
\]

Writing (3.37) in terms of the partitioning yields

\[
\ell_1 x_1 = b_1, \quad \ell_{21} x_1 + L_{22} x_2 = b_2.
\]

(3.38)

Clearly, \( x_1 = b_1 \ell_1^{-1} \) and (3.37) reduces to the linear system

\[
L_{22} x_2 = b_2 - \ell_{21} x_1.
\]

(3.39)

This is again a lower triangular system, and the component \( x_2 \) can be recovered from the first row of (3.39) in the same way that \( x_1 \) was recovered from the first row of (3.37). All of the component of \( x \) can be computed in the order \( x_1, \ldots, x_n \).

It is common practice to use the same vector to store \( x \) as we use to store \( b \), thus, in effect, performing the operation

\[
x \leftarrow L^{-1} b.
\]

(3.40)

An procedure to perform the operation (3.40), called forward substitution is next.
Function 3.2 (Forward Substitution)
%Perform the operation $x \leftarrow L^{-1}b$ for $b \in \mathbb{R}^n$
% and $L \in \mathbb{R}^{n \times n}$ lower triangular

```
function x = forsub(L, b)
    n = length(b);
    x = b;
    for i = 1 : n - 1
        x(i) = x(i)/L(i, i);
        x(i + 1 : n) = x(i + 1 : n) - L(i + 1 : n, i) * x(i);
    end
    x(n) = x(n)/L(n, n);
end
```
% forsub requires $n^2 + O(n)$ flops.

A square matrix $R \in \mathbb{R}^{n \times n}$ is upper triangular if
\[
    r_{ij} = 0 \quad i > j. \tag{3.41}
\]

Clearly, $R$ is upper triangular if $R = L^T$ where $L$ is lower triangular. It may be written
\[
    R = \begin{pmatrix}
    r_{11} & r_{12} & \cdots & r_{1n} \\
    0 & r_{22} & \cdots & r_{2n} \\
    \vdots & \vdots & \ddots & \vdots \\
    0 & 0 & \cdots & r_{nn}
    \end{pmatrix}.
\]

Clearly, $R$ is nonsingular if and only if $r_{ii} \neq 0, i = 1, \ldots, n$.

To solve
\[
    Rx = b \tag{3.42}
\]
consider the partitioning
\[
    R = \frac{1}{n-1} \begin{pmatrix} R_{11} & r_{12} \\ 0 & r_{nn} \end{pmatrix}, \quad x = \frac{1}{n-1} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}, \quad b = \frac{1}{n-1} \begin{pmatrix} b_1 \end{pmatrix}.
\]

Then (3.42) becomes
\[
    r_{nn}x_n = b_n, \quad R_{11}x_1 + r_{12}x_n = b_1.
\]

Thus $x_n = b_n r_{nn}^{-1}$ and the components of $x$ can be recovered in the order $x_n, x_{n-1}, \ldots, x_1$. 

Matrix Computational Concerns
3.2 Solution of Triangular Linear Systems

The procedure to perform the operation

\[ x \leftarrow R^{-1}b \]

is given next. The solution of (3.42) is called back substitution.

**Function 3.3 (Back Substitution)**

% Perform the operation \( x \leftarrow R^{-1}b \) for \( b \in \mathbb{R}^{n} \)
% and \( R \in \mathbb{R}^{n \times n} \) upper triangular

\[
\text{function } x = \text{backsub}(R, b) \\
\% \\
\% Back substitution with replacement. \\
\% For an upper triangular matrix \( R \), this performs the operation \\
\% \( x \leftarrow R^{-1}b \) \\
\%
\]

\[
n = \text{length}(b); \\
x = b; \\
\text{for } i = n : -1 : 2 \\
x(i) = x(i) / R(i, i); \\
x(1: i - 1) = x(1: i - 1) - R(1: i - 1, i) x(i); \\
\text{end for} \\
x(1) = x(1) / R(1, 1); \\
\text{end backsub} \\
\% \text{backsub} \text{ requires } n^2 / 2 + O(n) \text{ flops.}
\]

Both procedures ?? and 3.3 are numerically stable in the strongest possible sense. Higham [?, 2002, Chapter 8] gives a comprehensive discussion the stability results on solving triangular systems of equations. Below is one useful result [?, Theorem 8.5 p.142].

**Theorem 3.4** Let the lower (upper) triangular matrix \( T \in \mathbb{R}^{n \times n} \) be nonsingular and let the linear system \( T \hat{x} = b \) be solve by forward (back) substitution where the necessary sums are computed in any order. Then the computed solution \( \hat{x} \) satisfies

\[
(T + \delta T)\hat{x} = b, \quad |\delta T| \leq c(n)|T|\varepsilon_M + O(\varepsilon_M^2)
\]

where \( c(n) = O(n) \).

Theorem 3.4 states that back and forward substitution satisfy as strong a backward error bound as we could ever possibly expect.
3.3. Matrix Condition Numbers

The motivation for matrix condition numbers is expected accuracy in the solution of the linear system

\[ Ax = b \]  \hspace{1cm} (3.43)

where \( A \in \mathbb{R}^{n \times n}, \ x, b \in \mathbb{R}^n \), \( b \) is known and \( x \) is unknown.

We assume that either error in algorithms or data give us a perturbed system of the form

\[ (A + E)y = b + f. \]  \hspace{1cm} (3.44)

We wish to bound \( \|y - x\| \). We assume that \( A \) is nonsingular, thus that \( A^{-1} \) exists.

To begin, we give sufficient conditions for \( A + E \) to be nonsingular.

**Lemma 3.5** Suppose that \( A \in \mathbb{R}^{n \times n} \) is nonsingular, that \( \| \cdot \| \) denotes a vector norm and its associated induced matrix norm, and that \( \| A^{-1} E \| < 1 \). Then \( A + E \) is nonsingular and

\[ \| (A + E)^{-1} \| \leq \frac{\| A^{-1} \|}{1 - \| A^{-1} E \|}. \]

**Proof.** We show that \( (A + E)x = 0 \) if and only if \( x = 0 \). If \( (A + E)x = 0 \) then

\[ x = -A^{-1}E x. \]

Taking norms of both sides yields the inequality

\[ \| x \| \leq \| A^{-1}E \| \| x \|. \]

If \( \| x \| \neq 0 \), then

\[ \| x \| \leq \| A^{-1}E \| \| x \| < \| x \| \]

which is a contradiction. Thus \( x = 0 \) so that \( A + E \) is nonsingular.

If we let

\[ g = (A + E)x \]

for a non-zero vector \( y \) then

\[ A^{-1}g = (I + A^{-1}E)x. \]

The use of norm inequalities yields

\[ \| A^{-1} \|\|g\| \geq (1 - \| A^{-1}E \|)\|x\|. \]

Solving for \( \| x \| \) yields

\[ \frac{\| (A + E)^{-1} g \|}{\| g \|} \leq \frac{\| A^{-1} \|}{1 - \| A^{-1} E \|}. \]
Since \( \|g\| \) is arbitrary, the definition of induced matrix norms yields the result.

This lemma leads to a bound on difference between the solutions of the linear systems (3.43) and (3.44).

**Theorem 3.6**  Assume the hypothesis of Lemma 3.5. Then the solutions \( x \) of (3.43) and \( y \) of (3.44) satisfy

\[
\frac{\|y - x\|}{\|x\|} \leq \frac{\|A^{-1}E x\| + \|A^{-1}f\|}{(1 - \|A^{-1}E\|)\|x\|}.
\]

(3.45)

**Proof.** If we subtract (3.43) from (3.44) we obtain

\[ A(y - x) + Ey = f. \]

If we subtract \( Ex \) from both sides of the above equation we get

\[ (A + E)(y - x) = f - Ex. \]

Lemma 3.5 established that \( A + E \) is nonsingular, thus multiplying both sides by \( A^{-1} \) gives

\[ (I + A^{-1}E)(y - x) = A^{-1}f - A^{-1}Ex. \]

The use of norm inequalities yields

\[
(1 - \|A^{-1}E\|)\|y - x\| \leq \|A^{-1}f\| + \|A^{-1}Ex\|. \quad (3.46)
\]

Dividing both sides of (3.46) by \( (1 - \|A^{-1}E\|)\|x\| \) yields (3.45).

The use of this theorem depends upon what assumptions can be made about \( E \) and \( f \).

The simplest and probably most common assumption is that

\[
\|E\| \leq \epsilon\|A\|, \quad \|f\| \leq \epsilon\|b\| \quad (3.47)
\]

for some vector norm \( \| \cdot \| \) and its associated induced matrix norm. We can relax the assumption on \( b \) to

\[
\|f\| \leq \epsilon\|A\|\|x\|. \quad (3.48)
\]

The assumption (3.47) leads to

\[
\|A^{-1}E\| \leq \|A^{-1}\|\|E\| \leq \epsilon\|A^{-1}\|\|A\|,
\]

and

\[
\|A^{-1}f\| \leq \|A^{-1}\|\|f\| \leq \epsilon\|A^{-1}\|\|A\|\|x\|.
\]
If we define the condition number of $A$ relative to the norm $\| \cdot \|$ by
\[
\kappa(A) = \|A^{-1}\| \|A\|,
\]
then these inequalities are
\[
\|A^{-1}E\| \leq \varepsilon \kappa(A), \quad \|A^{-1}f\| \leq \varepsilon \kappa(A)\|x\|.
\]
Thus if $\varepsilon \kappa(A) < 1$, then Theorem 3.6 implies
\[
\frac{\|y - x\|}{\|x\|} \leq \varepsilon \frac{\kappa(A)}{1 - \varepsilon \kappa(A)} \tag{3.49}
\]
In the two-norm, $\kappa(A)$ has an elegant interpretation in terms of the SVD, namely that
\[
\kappa_2(A) = \|A^{-1}\|_2 \|A\|_2 = \sigma_1/\sigma_n.
\]
The condition number is used to characterize distance from singularity. We note that if
\[
\varepsilon \kappa(A) < 1
\]
then $\|A^{-1}E\| < 1$ and therefore $A + E$ is nonsingular.

In any induced norm $\kappa(A)$ characterizes the distance to singularity. We define
\[
\text{dist}_\text{sing}(A) = \min \left\{ \frac{\|E\|}{\|A\|} : A + E \text{ singular} \right\}.
\]
Gastinel [?; 1970] proved that
\[
\text{dist}_\text{sing}(A) = \frac{1}{\kappa(A)}
\]
in any of the $p$-norms. For the proof see [?; 2002, Theorem 6.5, p.111].

The norms for which this result is most important are the two-norm, one-norm and infinity-norm. For all three we can write an expression for a matrix $E$ such that
\[
A + E \text{ singular} , \quad \frac{\|E\|}{\|A\|} = \kappa(A)^{-1} \tag{3.50}
\]
For the two-norm, $E$ is given by
\[
E = -\psi_n u_n v_n^T
\]
where $(\psi_n, u_n, v_n)$ is the $n$th singular triplet of $A$.

If we let $j_{\text{max}}$ be an index such that
\[
\|A^{-1}\|_1 = \|A^{-1}e_{j_{\text{max}}}\|_1.
\]
Then
\[
E = -\gamma e_{j_{\text{max}}} c^T, \quad c = \text{sign}(A^{-1}e_{j_{\text{max}}}),
\]
3.3 Matrix Condition Numbers

\[ \gamma = 1 / \|A^{-1}\|_1 \]

satisfies (3.50) for the one-norm.

For the infinity norm, just apply the one-norm formula to \(A^T\). Thus, if

\[ \gamma^{-1} = \|A^{-1}\|_{\infty} = \|A^{-T}\|_1 = \|A^{-T}e_{j_{\text{max}}}\|_1. \]

then

\[ E = -\gamma ce_{j_{\text{max}}}^T, \quad c = \text{sign}(A^{-T}e_{j_{\text{max}}}) \]

satisfies (3.50) for the infinity-norm.

With this characterization, we think of a matrix as singular to machine precision (relative to a norm) if

\[ \varepsilon_M \kappa(A) \geq 1 \]

for some norm. In later sections, we will give methods for estimating \( \kappa(A) \) in the two-norm, one-norm, and infinity-norm.

Another popular assumption for \( E \) and \( f \) is to assume that they are componentwise bounded. That is,

\[ |E| \leq \varepsilon |A|, \quad |f| \leq \varepsilon |b| \leq \varepsilon |A||x|. \]

\[ \|A^{-1}E\| \leq \varepsilon \|A^{-1}\|_1 \|A\|, \quad \|A^{-1}f\| \leq \varepsilon \|A^{-1}\|_1 \|A\| \|x\|. \]

For instance, solution of triangular systems of linear systems satify this assumption. Thus we define the condition number

\[ \text{cond}(A) = \|A^{-1}\|_1 \|A\|, \]

and obtain the bound

\[ \frac{\|y - x\|}{\|x\|} \leq 2\varepsilon \frac{\text{cond}(A)}{1 - \varepsilon \text{cond}(A)}. \]

For both the one-norm and the infinity-norm it is easily verified that

\[ \text{cond}_1(A) = \|A^{-1}\|_1 \|A\| \leq \kappa_1(A), \]

\[ \text{cond}_\infty(A) = \|A^{-1}\|_\infty \|A\| \leq \kappa_\infty(A). \]

Unfortunately, these inequalities do not hold for the two-norm. Our best bound would be

\[ \text{cond}_2(A) = \|A^{-1}\|_2 \leq \min\{\kappa_\infty(A) \kappa_1(A) \}^{1/2}, \kappa_F(A) \} \leq n \kappa_2(A). \]

The componentwise condition number does not lead to a simple characterization of distance to singularity (see Rohn [?, 1983] and Demmel [?, 1992]).
3.4. Efficient Implementation of Matrix Operations

When writing software for matrix operations, the two principal considerations are accuracy and speed. We noted in §3.1, the accuracy of a particular computation can depend upon how it is coded, the same is true of speed.

Matrix and vector operations are often placed into three separate categories called levels:
1. Vector operations (level 1),
2. Matrix-vector operations (level 2),
3. Matrix-matrix operations (level 3).

Level 1 operations are $O(n)$ operations on $O(n)$ data, level 2 operations are $O(n^2)$ operations on $O(n^2)$ data, and level 3 operations are $O(n^3)$ operations on $O(n^2)$ data.

The level terminology came about in the design of the Basic Linear Algebra Subprograms (BLAS), a set of FORTRAN callable routines. A C interface to the BLAS and a Java version of the BLAS are available. The level 1 BLAS (vector operations only) were introduced by Lawson, Hanson, Kincaid, and Krogh [26, 1979]. Subsequently, level 2 BLAS (matrix-vector operations) were introduced by Dongarra, DuCroz, Hammarling, and Hanson [14, 1988], and level 3 BLAS were introduced by Dongarra, DuCroz, Hammarling, and Duff [13, 1990].

For details on the BLAS, read the original papers [26, 1979], [14, 1988], [13, 1990], or the excellent summary in Freeman and Phillips [15, 1992]. Freeman and Phillips also explain the role of the BLAS in parallel and distributed computing. The BLAS undergo continuing development; the BLAS web page http://www.netlib.org/blas/ gives the current state of the BLAS. See also the articles [11, 2002], [12, 2002], and [4, 2002].

When coding matrix algorithms that require $O(n^3)$ or more operations, it is advisable to make the maximum use of level 3 operations. The following example illustrates why.

Consider the coding of the matrix multiply and add operation

$$ C \leftarrow C + AB $$

(3.51)

where $A, B, C \in \mathbb{R}^{n \times n}$. We have made all of the matrices square to keep the explanation simple, but the following discussion applies with minor modifications to any case where the operation (3.51) is defined.

Componentwise, the operation (3.51) is

$$ c_{ij} \leftarrow c_{ij} + \sum_{k=1}^{n} a_{ik} b_{kj}, \quad i, j = 1, \ldots, n. $$

(3.52)

This is just the double nested loop
3.4 Efficient Implementation of Matrix Operations

Function 3.7 function 

\[
C = \text{matmul}(A, B)
\]

\[
n = \text{length}(A); \quad \% \text{For simplicity we assume that } A \text{ and } B \text{ are square.}
\]

\[
\text{for } j = 1:n
\]

\[
\text{for } i = 1:n
\]

\[
\text{for } k = 1:n
\]

\[
c_{ij} \leftarrow c_{ij} + a_{ik} b_{kj}
\]

\[
\text{end for}
\]

\[
\text{end for}
\]

\[
\text{end for.}
\]

\[
\text{end matmul}
\]

This computation requires \(n^3\) additions and \(n^3\) additions, but that does not measure the true cost of Procedure 3.7. Most of the cost is in memory accesses, moving elements of \(A, B,\) and \(C\) in and out of storage.

Computers have memory arranged in a hierarchy, arranged from fastest memory with the least capacity to slowest memory with most capacity. At the top of the hierarchy are \(fast\) \(registers\) attached to arithmetic processors. These registers can store only a few values at a time, but all arithmetic is done here. In the middle of the hierarchy is \(cache\) or \(local\) \(memory.\) This is a small memory with a fast access time. In many computers, there are actually several caches arranged in their own hierarchy, but for our discussion, we will assume there is only one cache. At the bottom of the hierarchy, there is \(global\) \(memory\) which has a large storage capacity, but slow access time. There are also levels below global memory which include disks and tapes which have much slower access time. The memory hierarchy is shown in Figure ??.

Accesses to global memory are almost always the dominant cost of computation. In fact, the gap between access time for global memory access and cache access is increasing with improving technology. Thus, in our discussion, we will only count accesses to global memory.

We will assume that our cache memory can hold at least \(3n\) elements but that \(n^2\) elements far exceed its capacity. Thus the cache can easily hold a row or column from each of \(A, B,\) and \(C,\) but not an entire matrix.

If we were to run Procedure 3.7, a good compiler would recognize that the \(jth\) column of \(B\) can be placed into cache between the \(i\) and \(j\) loop, and the \(ith\) row of \(A\) the \((i,j)th\) entry of \(C\) can be placed into cache between the \(i\) and \(k\) loop. Also, the \(k\) loop is just a dot product between row \(j\) of \(B\) and column \(i\) of \(A,\) which is level 1 operation. Thus we can rewrite Procedure 3.7 as

Function 3.8 function \(C = \text{matmul}(A, B)\)
$n = \text{length}(A)$;
for $j = 1 : n$
    Place $B(:,j)$ into cache.
    for $i = 1 : n$
        Place $A(i,:)$ and $c_{ij}$ into cache.
        $c_{ij} \leftarrow c_{ij} + A(i,:) \ast B(:,j)$;
    end for
end for.
end \text{matmul}$

Many modern machines have special vector registers that can perform a dot product very quickly, so there is some speedup in replacing the inner loop with a dot product. Note that every element of $A$ is read out of storage $n$ times, which requires $n^3$ storage accesses. Fortunately, each element of $B$ and $C$ requires only one access to global memory. Thus Procedure 3.8 requires $n^3 + 2n^2$ accesses to global memory.

We can make this program somewhat more efficient by replacing the $k$ loop with a Gaxpy operation.

\textbf{Function 3.9 function} $C = \text{matmul2} (A,B)$

\begin{verbatim}
 n = \text{length}(A);
 for \ j = 1 : n
   Place $C(:,j)$ and $B(:,j)$ into cache.
   $C(:,j) \leftarrow C(:,j) + A \ast B(:,j)$
 end for
end \text{matmul2}$

If the columns $C(:,j)$ and $B(:,j)$ are loaded into cache, then $B$ and $C$ each require only $n^2$ accesses to global memory. However, the matrix $A$ is too large to be put into cache, so we must reaccess it from global memory each time through the loop. Thus we must still do $n^3 + 2n^2$ accesses to global memory. Some machines do matrix-vector multiplication very efficiently and thus, on these machines, Procedure 3.9 could be more efficient than Procedure 3.8. However, there will little difference in the amount of storage access time. Moreover, it is reasonable to expect that a good compiler could take code such as Procedure 3.7 and change it to 3.9 or some level 2 code.
Let $n = pq$ for some integers $p$ and $q$. Let $A$ have the block partitioning

$$
A = \begin{pmatrix}
A_{11} & A_{12} & \cdots & A_{1p} \\
A_{21} & A_{22} & \cdots & A_{2p} \\
\vdots & \vdots & \ddots & \vdots \\
A_{p1} & A_{p2} & \cdots & A_{pp}
\end{pmatrix}
$$

(3.53)

where each $A_{ij} \in \mathbf{R}^{q \times q}$. Let $B$ and $C$ be partitioned conformally. In programming terms $A_{ij} = A(1 + (i - 1) * q : i * q, 1 + (j - 1) * q : j * q)$. Then the matrix multiply and add in (3.51) can be written

**Function 3.10** function $C = \text{matmul-blocks} (A, B)$

$n = length(A)$; % Let $p$ and $q$ be integers such that $n = pq$

$\text{jlow} \leftarrow 1; \text{jhigh} \leftarrow q$;

for $j = 1:p$

$\text{klow} \leftarrow 1; \text{khight} \leftarrow q$;

for $k = 1:p$

$\text{ilow} \leftarrow 1; \text{ihight} \leftarrow q$;

for $i = 1:p$

Place $C(\text{ilow}; \text{ihight}, \text{jlow}; \text{jhigh})$, $A(\text{ilow}; \text{ihight}, \text{klow}; \text{khight})$, and $B(\text{klow}; \text{khight}, \text{jlow}; \text{jhigh})$ into cache.

$C(\text{ilow}; \text{ihight}, \text{jlow}; \text{jhigh}) \leftarrow C(\text{ilow}; \text{ihight}, \text{jlow}; \text{jhigh}) + A(\text{ilow}; \text{ihight}, \text{klow}; \text{khight}) * B(\text{klow}; \text{khight}, \text{jlow}; \text{jhigh})$

$C_{ij} \leftarrow C_{ij} + A_{ik}B_{kj}$

$\text{ilow} \leftarrow \text{ihight} + 1; \text{ihight} \leftarrow \text{ihight} + q$;

end for

$\text{klow} \leftarrow \text{khight} + 1; \text{khight} \leftarrow \text{khight} + q$;

end for

$\text{jlow} \leftarrow \text{jhigh} + 1; \text{jhigh} \leftarrow \text{jhigh} + q$;

end for

end matmul-blocks

Assume that $q = \alpha \sqrt{n}$ where $\alpha \leq 1$ but $\alpha$ is bounded away from 0. Then $p = \beta \sqrt{n}$ where $\beta = 1/\alpha$. In the inner loop, the block $C_{ij}$, $A_{ik}$ and $B_{kj}$ have no more than $3\alpha^2 n$ elements and therefore fits nicely into the cache. There are $3q^2$ storage accesses each time through the inner loop which comes to $3p^3q^2$ accesses for the entire procedure. Since $n = pq$, this is $3\beta n^{2.5}$ storage accesses. Since $\beta$ is constant this is an order of magnitude savings over the `matmul`, `matmulI`, and `matmul2` codes. That is because the inner loop of `matmul-blocks` does $q^2$ storage access, but $O(q^3)$ operations. Hong and Kung [18, 1981] show that $O(n^3/p)$ memory accesses is a lower bound on the implementation of this matrix multiplication routine.
The cache size cannot be expected to be $O(\sqrt{n})$. However, it is appropriate to choose the blocksize so that the blocks in the inner loop of Procedure 3.10 fit into the cache with little room to spare. The only reasonable way to determine the appropriate blocksize $q$ is by computational experiment.

Of course, most programmers do not want to be bothered with running a computational experiment to determine the best value of $q$ for a particular computer, especially if the results are useless when the computer is replaced. Instead, we prefer to have a standard matrix multiply and add supplied by the manufacturer that is optimized for the particular computer. That is, in fact, how a manufacturer supplied matrix multiplication routine should be implemented.

Algorithms such as coded by Procedure 3.10 are called block algorithms. For matrix operations requiring $O(n^3)$ operations or more, properly implemented block algorithms will often greatly reduce the number of accesses to main memory, resulting in much more efficient code. In the design of computer algorithms, reducing the number of storage access is at least as important and often much more important than reducing the number of operations.

3.5. Simple Computations are Not Always So Simple

We give two examples of simple computations that give rise to interesting issues when implemented in floating point arithmetic.

For both examples, we start with a vector $\mathbf{x}$. Since as stated in §3.4, memory accesses consume far more time than arithmetic operations, we want to access $\mathbf{x}$ from storage only once. Memory accesses are much harder to measure than operation counts, but for these two examples it is simple.

For the first example, computing the sample variance, we show that simply rewriting the formula can produce three different algorithms that are respectively strongly forward stable, unstable, and backward stable.

The second example shows that an algorithm for computing the two-norm that minimizes memory accesses and is resistant to underflow and overflow must be designed carefully.

3.5.1. Computation of Sample Means and Variances

Let $\mathbf{x} = (x_1, \ldots, x_n)^T$ be a random vector. The sample mean $\bar{x}$ follows the formula

$$\bar{x} = \frac{1}{n} \sum_{i=1}^{n} x_i. \quad (3.54)$$

In floating point arithmetic we actually compute

$$fl(\bar{x}) = \sum_{i=1}^{n} x_i(1 + \eta_i)$$
where

\[
|\eta_1| \leq n \varepsilon_M + O(\varepsilon_M^2) \\
|\eta_k| \leq (n - i + 2)\varepsilon_M + O(\varepsilon_M^2) \quad i = 1, 2, \cdots, n.
\]

If we use pairwise summation

\[
|\eta_k| \leq ([\log_2 n] + 1)\varepsilon_M + O(\varepsilon_M^2). \tag{3.55}
\]

The sample variance of the random sample \(\{x_i\}_{i=1}^n\) is given by the formula

\[
s^2 = \frac{1}{n - 1} \sum_{i=1}^{n} (x_i - \bar{x})^2.
\]

The obvious algorithm to perform this computation simply follows the formula. One computes the mean by standard or pairwise summation and then computes the sum

\[
t_1 = (x_1 - \bar{x})^2 \tag{3.56} \\
t_k = t_{k-1} + (t_k - \bar{x})^2 \quad k = 2, \cdots, n \tag{3.57} \\
s^2 = \frac{1}{n - 1} t_n. \tag{3.58}
\]

Chan, Golub, and LeVeque [6, 1983] call (3.56)–(3.58) the standard two-pass algorithm. That is because it requires us to access the data twice: once to compute \(\bar{x}\) and again to compute \(s^2\). Clearly, pairwise summation could also be used to compute \(t_n\).

Chan and Lewis [7, 1979] derived a condition number with respect to a data set \(\{x_i\}_{i=1}^n\) for the computation of \(s^2\). We use a slightly less restrictive version of condition number given by Barlow [3, 1991].

Suppose that we introduce a relative perturbation of size \(\xi\). That is, let

\[
x(\xi) = x + \xi v
\]

where \(v = (v_1, v_2, \ldots, v_n)^T\) is a vector that satisfies \(\|v\|_2 = \|x\|_2\). We now show how this “small change” in \(x\) will affect \(s^2\). Define

\[
s^2(\xi) = \frac{1}{n - 1} \sum_{i=1}^{n} (x_i + \xi v_i - \bar{x}(\xi))^2 \tag{3.59}
\]

where

\[
\bar{x}(\xi) = \frac{1}{n} \sum_{i=1}^{n} (x_i + \xi v_i) = \bar{x} + \xi \bar{v}. \tag{3.60}
\]
Let $c = (1, 1, \ldots, 1)^T \in \mathbb{R}^n$. Then in norm terms

$$s(\xi) = \frac{1}{\sqrt{n-1}}\|x(\xi) - \bar{x}c\|_2. \quad (3.61)$$

That may be written

$$s(\xi) = \frac{1}{\sqrt{n-1}}\|x - \bar{x}c + \xi(v - \bar{v}c)\|_2.$$

The use of the triangle inequality gives us

$$s(\xi) \leq \frac{1}{\sqrt{n-1}}(\|x - \bar{x}c\|_2 + \xi\|v - \bar{v}c\|_2) = s + \xi\|v - \bar{v}c\|_2\sqrt{n-1}.$$ 

Reversing the roles of $s$ and $s(\xi)$ yields

$$s(\xi) \geq s - \xi\|v - \bar{v}c\|_2\sqrt{n-1}. $$

Thus

$$\frac{|s(\xi) - s|}{s} \leq \frac{\xi\|v - \bar{v}c\|_2}{\sqrt{n-1}s}. \quad (3.62)$$

First note that

$$\|v - \bar{v}c\|_2^2 = \|v\|_2^2 - 2\bar{v}c^Tv + \bar{v}^2\|c\|_2^2.$$ 

Since $c$ is a vector of ones, $c^Tv = n\bar{v}$ and $\|c\|_2^2 = n$, thus

$$\|v - \bar{v}c\|_2^2 = \|v\|_2^2 - n\bar{v}^2 \leq \|v\|_2^2 = \|x\|_2^2. \quad (3.63)$$

For $v$, we can say,

$$\frac{|s(\xi) - s|}{s} \leq \xi\kappa_s(x), \quad (3.64)$$

where

$$\kappa_s(x) = \frac{\|x\|_2}{\sqrt{n-1}s}. \quad (3.65)$$

The bound (3.64) can be achieved by judicious choice of $v$. Thus a relative error of magnitude $\varepsilon_M\kappa_s(x)$ in $s$ is possible from just representation errors on the computer.

An algorithm to compute $s$ is weakly forward stable if it produces a computed $\tilde{s}$ that satisfies

$$\frac{\tilde{s} - s}{s} \leq \varepsilon_M\kappa_s(x) + O(\varepsilon_M^2) \quad (3.66)$$
where \( g(n) \) is a modestly sized function of \( n \). However, choosing \( \xi = \varepsilon_M g(n) \), we note that there is always a vector \( \mathbf{x}(\xi) = \mathbf{x} + \xi \mathbf{v} \) such that \( \mathbf{s} = s(\xi) \) satisfies (3.66). Thus, for this problem, a bound of the form (3.66) is sufficient to show that an algorithm is backward stable.

Chan, Golub, and LeVeque [6, 1983] showed that the two-pass algorithm does better than (3.66). It satisfies a bound of the form

\[
\frac{\|\mathbf{s} - s\|}{s} \leq \varepsilon_M g(n) + \varepsilon_M^2 [g(n)]^2 \kappa_n(x)^2 + O(\varepsilon_M^3) \tag{3.67}
\]

where

\[
g(n) = \begin{cases} 
O(n) & \text{if standard summation is used,} \\
O(\log n) & \text{if pairwise summation is used}.
\end{cases} \tag{3.68}
\]

This is a statement that, except for an error of \( O(\varepsilon_M^2) \), the standard two-pass algorithm is strongly forward stable.

We will now prove that bound for standard summation, the proof is identical for pairwise summation.

First, the bounds on floating point sums assure us that

\[
\text{fl}(\bar{x}) = \bar{x} + \eta_1 \|\mathbf{x}\|_1 \tag{3.69}
\]

where

\[
|\eta_1| \leq \varepsilon_M + O(\varepsilon_M^2)
\]

Thus

\[
\text{fl}(s^2) = \text{fl}(\frac{1}{n-1}\|\mathbf{x} - \text{fl}(\bar{x})\mathbf{c}\|_2^2) = \frac{1}{n-1}(\|\mathbf{x} - \text{fl}(\bar{x})\mathbf{c}\|_2^2 + \eta_2 \|\mathbf{x} - \text{fl}(\bar{x})\mathbf{c}\|_2^2) \tag{3.70}
\]

where

\[
|\eta_2| \leq (n + 3)\varepsilon_M + O(\varepsilon_M^2).
\]

Using (3.57) in \( \mathbf{x} \) yields

\[
\|\mathbf{x} - \text{fl}(\bar{x})\mathbf{c}\|_2^2 = \|\mathbf{x} - \bar{x}\mathbf{c}\|_2^2 + 2(\text{fl}(\bar{x}) - \bar{x})\bar{x}^T(\mathbf{x} - \bar{x}\mathbf{c}) + (\text{fl}(\bar{x}) - \bar{x})^2\|\mathbf{c}\|_2^2. \tag{3.71}
\]

Since \( \mathbf{c}^T(\mathbf{x} - \bar{x}\mathbf{c}) = 0 \), we combine (3.71) with (3.69) to become

\[
\|\mathbf{x} - \text{fl}(\bar{x})\mathbf{c}\|_2^2 - \|\mathbf{x} - \bar{x}\mathbf{c}\|_2^2 \leq \eta_1^2\|\mathbf{x}\|_1^2\|\mathbf{c}\|_2^2 \leq n^2\eta_2^2\|\mathbf{x}\|_2^2. \tag{3.72}
\]

Thus the rounding error in the mean has only a second order effect on the the variance. Moreover, substituting \( \bar{x} \) for \( \text{fl}(\bar{x}) \) in into (3.70) has an effect of \( O(\varepsilon_M^3) \). Thus

\[
|\text{fl}(s^2) - s^2| \leq \varepsilon_M(n + 3)s^2 + \varepsilon_M^2n^2\|\mathbf{x}\|_2^2 + O(\varepsilon_M^3)
\]
which becomes
\[
\frac{\alpha(s^2) - s^2}{s^2} \leq \varepsilon_M(n + 3) + \varepsilon_M^2 n^2 \kappa_s^2(\mathbf{x}) + O(\varepsilon_M^3). \tag{3.73}
\]
If we makes use of the fact that for small values of \(\eta\),
\[
\sqrt{s^2(1 + \eta)} = |s|(1 + 0.5\eta + O(\eta^2)),
\]
we obtain a bound of the form (3.67).

Note that perturbation of the mean has only a second order effect on the variance. Once the mean is substracted from the data, it is as though the algorithm is performed on \(\mathbf{x} - \bar{x}\mathbf{e}\) instead of \(\mathbf{x}\). For that problem,
\[
\kappa_s(\mathbf{x} - \bar{x}\mathbf{e}) = \frac{||\mathbf{x} - \bar{x}\mathbf{e}||_2}{\sqrt{n - 1}s} = 1.
\]

Thus, the original problem is mapped onto a well-conditioned problem with only a second order error.

Since many modern computers have hierarchical memories, where the fastest memory, the cache, is limited in size, the algorithm (3.56)-(3.58) may be undesirable. If the sample is too large to be stored in the cache, it could be slow.

A common practice is to manipulate the definition of \(s^2\) into
\[
s^2 = \frac{1}{n-1} \sum_{i=1}^{n} x_i^2 - \frac{n}{n-1} \bar{x}^2. \tag{3.74}
\]

The form (3.74) leads to another algorithm to compute the sample variance. It is given by
\[
m_1 = x_1, t_1 = m_1^2, \tag{3.75}
\]
\[
m_{k+1} = m_k + x_{k+1}, \quad t_{k+1} = t_k + x_{k+1}^2, \quad k = 1, 2, \ldots, n - 1, \tag{3.76}
\]
\[
\bar{x} = \frac{1}{n} m_n, \tag{3.77}
\]
\[
s^2 = \frac{1}{n-1} t_n - \frac{n}{n-1} \bar{x}^2. \tag{3.78}
\]

Again following [6, 1983], we call this the standard one-pass algorithm. It can be computed by accessing the data only once. Thus it would be faster on large samples that cannot be placed in fast memory and amenable for pairwise summation.

Unfortunately, the one-pass algorithm is not stable; it satisfies no bound of the form (3.66). The best bound that can be obtained for the relative error in the computed value \(\hat{s}^2\) is
\[
\frac{|\hat{s} - s|}{s} \leq \varepsilon_M g(n) \kappa_s(\mathbf{x})^2 + O(\varepsilon_M^2), \tag{3.79}
\]
3.5 Simple Computations are Not Always So Simple

where $g(n)$ satisfies (3.68).

To avoid this problem, several authors, Youngs and Cramer[33, 1971],
West[31, 1979], and E.H. Hansen [17, 1975] derived stable algorithms for com-
puting the sample variance using only one pass through the data. In [6, 1983],
it was pointed out that all of these algorithms were essentially equivalent. We
now give the version due to West [31, 1979].

$$m_1 = x_1, \quad t_0 = 0,$$

$$m_{k+1} = m_k + (x_{k+1} - m_k)/(k + 1), \quad k = 1, 2, \ldots, n - 1,$$

$$t_{k+1} = t_k + \frac{k - 1}{k} (x_k - m_{k-1})^2, \quad k = 1, 2, \ldots, n - 1,$$

$$\bar{x} = \frac{1}{n} m_n, \quad s^2 = \frac{1}{n - 1} t_n. \quad (3.83)$$

Chan and Lewis [7, 1979] showed that West’s algorithm satisfied a bound
of the form (3.66) where

$$g(n) = O(n). \quad (3.84)$$

This algorithm satisfies all of our needs except two: (1) it is not amenable to
pairwise summation; (2) it allows us to add only one new sample at a time.
In many applications it is necessary to combine samples.

The following algorithm was proposed by Chan, Golub, and LeVeque [6,
1983] to resolve these issues. For $1 \leq i \leq j \leq n$, let

$$m_{ij} = \sum_{k=i}^{j} x_k,$$

and let

$$t_{ij} = \sum_{k=i}^{j} (x_k - \frac{1}{j - i} m_{ij})^2. \quad (3.86)$$

Clearly,

$$s^2 = \frac{1}{n - 1} t_{1,n}. \quad (3.87)$$

We can compute the sample variance using the updating procedure

$$t_{1,m+\ell} = t_{1,m} + t_{m+1,m+\ell} + \frac{m}{\ell (m + \ell)} (\frac{\ell}{m} m_{1,m} - m_{m+1,m+\ell})^2. \quad (3.88)$$

When $m = \ell$, this reduces to

$$t_{1,2m} = t_{1,m} + t_{m+1,2m} + \frac{1}{2m} (m_{1,m} - m_{m+1,2m})^2. \quad (3.89)$$
By choosing $m = \left\lfloor \frac{n}{3} \right\rfloor$ and $l = \left\lfloor \frac{n}{2} \right\rfloor$ and using (3.86)–(3.89) recursively, we can compute the sample variance in $\left\lfloor \log_2 n \right\rfloor$ levels of recursion. This procedure is also appropriate for computation on distributed computers [3, 1991 6, 1983].

Barlow [3, 1991] proved that, in floating arithmetic, such a procedure would compute a sample variance $\hat{s}$ such that

$$\frac{|\hat{s} - s|}{s} \leq \varepsilon_M g(n) \kappa_s(x) + O(\varepsilon_M^2)$$

(3.90)

where $g(n) = O(\log n)^2$.

None of the one-pass algorithms transform the data into a better conditioned problem, thus the condition number of the sample variance problem is realistic for these algorithms. Thus West’s algorithm and Chan, Golub, and LeVeque’s algorithm are backward stable, but not strongly forward stable.

3.5.2. Computing the Two-Norm

As was stated in §3.1.3, we would expect an algorithm to compute the two-norm in floating point arithmetic to produce the correct two-norm of any machine representable vector $x$ to relative accuracy as long as $||x||_2$ does not overflow.

The algorithm of simply using the formula

$$||x||_2 = \left( \sum_{i=1}^{n} x_i^2 \right)^{1/2}$$

is unsatisfactory because $x_i^2$ may overflow or underflow even when $||x||_2$ is in range.

This can be seen for $n = 2$. For instance, suppose we are working on an IEEE Standard 754 computer where the smallest positive computer number is

$$\omega = 2^{-154}.$$  

(3.91)

Let $x \in \mathbb{R}^2$ be specified by

$$x = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 3 \times 2^{-79} \\ 4 \times 2^{-79} \end{pmatrix}.$$ 

The direct use of the formula would start by squaring $x_1$ and $x_2$ and this would yield

$$\text{fl}(x_1^2) = \text{fl}(9 \times 2^{-158}) = 0$$

since $9 \times 2^{-158} < \omega$ while

$$\text{fl}(x_2^2) = \text{fl}(16 \times 2^{-158}) = 2^{-154} = \omega.$$
3.5 Simple Computations are Not Always So Simple

After summing and taking square roots we get
\[ f(||x||_2) = 2^{-77} = 4 \times 2^{-79} \]  
(3.92)
when the correct answer is \( 5 \times 2^{-79} \). The result here is not because of a rounding error, it is because an underflow that is easily avoided.

The correct way to compute \( ||x||_2 \) for this case would be to compute is to use the routine \texttt{twonorm2} below.

**Function 3.11 (Two Norm of a Two-Vector)**

```plaintext
function norm2 = twonorm2(x)
    absx1 = abs(x_1); absx2 = abs(x_2);
    if absx1 >= absx2
        norm2 = absx1 * sqrt(1 + (absx2/absx1)^2);
    else
        norm2 = absx2 * sqrt(1 + (absx1/absx2)^2);
    end
end twonorm2
```

The reader may verify that this procedure will obtain the correct result.

For \( x \in \mathbb{R}^n \), the above procedure generalizes to the two steps
\[
\gamma \leftarrow ||x||_\infty \\
||x||_2 \leftarrow \gamma \left( \sum_{i=1}^{n} (x_i/\gamma)^2 \right)^{1/2}.
\]
(3.93)  
(3.94)

The algorithm (3.93)–(3.94) computes \( ||x||_2 \) to relative accuracy as long as the answer is in the range of the computer. There may be underflows in the sum in (3.94), but they would have no more effect on the result than rounding.

If \( n \) is very large, the algorithm (3.93)–(3.94) could very slow. It requires two memory accesses for every component of \( x \), one the for the computation (3.93) and one for the computation (3.94).

Blue [5, 1978] proposed an algorithm for computing \( ||x||_2 \) that is as resistant to overflow as (3.93)-(3.94), but requires only one pass through the data. It separates the computation of \( ||x||_2 \) into three sums over the large, medium, and small portions of the vector. It requires some parameters from the floating point number system. It requires fewer square roots than the routine given below, but it is more complicated.

The routine we give below is based upon the LAPACK [1, p.137] routine \texttt{SLASSQ}. That routine actually returns two numbers, \( ||x||_\infty \) and \( ||x||_2/||x||_\infty \),
in case, \( \|x\|_2 \) overflows when \( \|x\|_\infty \) does not. Our function \texttt{slasq} will do
that computation, but our function \texttt{twonorm} will insist on returning \( \|x\|_2 \).
It just repeatedly applies \texttt{twonorm2} as below.

\textbf{Function 3.12 (Two Norm of a Two-Vector)}

\begin{verbatim}
function norm2 = twonorm(x)
    [inf_norm, ratio] = slasq(x);
    norm2 = ratio * inf_norm;
end twonorm

function [inf_norm, ratio] = slasq(x, inf_norm, ratio)
    n = length(x);
    inf_norm = |x_1|;
    ratio = 1;
    for k = 2:n
        abszk = |x_k|;
        if abszk > inf_norm
            factor = inf_norm/abszk;
            ratio = twonorm2([ratio * factor, 1])^T;
            inf_norm = abszk;
        else
            factor = abszk/inf_norm;
            ratio = twonorm2([ratio, factor])^T;
        end
    end
end slasq
\end{verbatim}

The above function does not resolve all issues that can occur in computing
the two-norm. It does obtain a result satisfying

\[
\hat{f}(\|x\|_2) = \|x\|_2(1 + \eta), \quad |\eta| \leq g(n)\varepsilon_M + O(\varepsilon_M^3) \quad (3.95)
\]

where \( g(n) = O(n) \).

If \( n \) is large, this bound is not satisfactory. For instance, consider a vector
of the form

\[ x = (1, \xi, \xi, \ldots, \xi)^T \in \mathbb{R}^n \]

where \( \sqrt{\omega} < \xi^2 < \varepsilon_M \) but \( n\xi^2 = O(1) \). If, say, \( n\xi^2 = 1.25 \), then \( \|x\|_2 = 1.5 \),
Function 3.12 will give the value 1. Note that (3.95) is satisfied, so, by our
definition, we have stable algorithm, but \( n \) is so large that the statement (3.95)
is meaningless. Several possible fixes have been proposed. The simplest is to
use Dekker’s [9, 1971] procedures for simulating a doubled precision adds and
multiples. Unfortunately, these procedures require a subroutine call for each add and multiply.
Bibliography


