Elements of Floating-point Arithmetic

Sanzheng Qiao Yimin Wei

August 2007

Chapter 1

Floating-point Numbers

1.1 Representations

On paper	In storage
$\pm d_1.d_2...d_t \times \beta^e$	stored in three fields,
$0 < d_1 < \beta$, $0 \leq d_i < \beta$ $(i > 1)$	usually consecutive:
β: base (or radix)	sign: 1 bit
e: exponent	exponent: store e
t: precision	fraction: store $d_1, ..., d_t$

The base β is almost universally 2. In this case, d_1 is always 1. Thus, it can be implicit, saving one bit, called *hidden bit*. Other commonly used bases are 10 and 16. Base 16 provides wider range, but wastes bits in fraction. For example, in base 16, when $1.0_{16} \times 16^0$ is stored in binary bits, there are three leading zeros. In general, there can be up to $\log \beta - 1$ leading zeros.

A floating-point number system is characterized by four parameters:

- base β (also called radix)
- precision t
- exponent range $e_{\min} \leq e \leq e_{\max}$

Machine precision, ϵ_M, is defined as the distance between 1.0 and the next larger floating-point number, that is, $\epsilon_M = \beta^{1-t}$.

Floating-point numbers are used to approximate real numbers, $\mathrm{fl}(x)$ denotes the floating-point number that approximates a real number x.

One way of measuring the error in an approximation is related to the absolute error. If the nearest rounding is applied and $\mathrm{fl}(x) = d_1.d_2 \cdots d_t \times \beta^e$, then the absolute error $|x - \mathrm{fl}(x)|$ is no larger than $1/2\beta^{-t+1}\beta^e$. Thus we define one *ulp* (unit in the last place) of a floating-point number $d_1.d_2 \cdots d_t \times \beta^e$ as $\beta^{-t+1}\beta^e$.

Another way of measuring the error in an approximation is related to the relative error. Again, if the nearest rounding is applied, then the relative error $|x - \mathrm{fl}(x)|/|x|$ is no larger than $1/2\beta^{-t+1}$, defined as the *unit of roundoff* denoted by u. When $\beta = 2$, $u = 2^{-t}$.

To illustrate the difference between ulp and u, we consider the real number 31/64. When $\beta = 2$ and $t = 4$, it is approximated by 1.000×2^{-1}. The relative error is bounded above by $2^{-4} = u$, while the absolute error is $2^{-6} = 1/4$ ulp, instead of 1/2 ulp, of 1.000×2^{-1}. The reason for wobbling by a factor of β is that when $31/64 = 1.1111 \times 2^{-2}$ is rounded to 1.000×2^{-1}, the exponent is increased by one.

Usually, we are only concerned about the magnitude of rounding error. Then, ulps and u can be used interchangeably. If the error in a floating-point number is n ulps or nu, then the number of contaminated digits is $\log_\beta n$. Also, since $\epsilon_M = 2u$, the machine precision ϵ_M can also be used in place of u or ulps.

1.2 A Small System

To simplify our presentation, we will often use a contrived small floating-point number system characterized by

β	t	$e_{\min}$	$e_{\max}$	Exponent width	Format width
2	4	-6	$+7$	4 bits	8 bits

From this small system, we can see that the floating-point numbers in each interval $[\beta^e, \beta^{e+1})$, $e_{\min} \le e \le e_{\max}$, are equally spaced, with the distance $\beta^{-t+1}\beta^e = 1$ ulp.

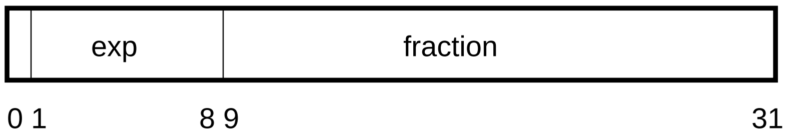
Example The decimal number 0.1 cannot be represented exactly in the small system, it is approximated by its nearest floating-point number 1.101×2^{-4}. The relative error is no larger than $u = 2^{-4}$ and the absolute error is no larger than $2^{-8} = 1/2$ ulp.

1.3 IEEE Floating-point Standard

IEEE 754 is a binary standard that requires base $\beta = 2$ and

	single precision	double precision
precision t	24	53
$e_{\min}$	-126	-1022
$e_{\max}$	127	1023

Single precision word format:



Precision
Double precision supports single, so that intermediate results in evaluating an expression are computed in high precision. Why? Extra-precise arithmetic attenuates the risk of error due to roundoff. For example, in evaluating an inner product of two single precision vectors, the intermediate results are computed in double precision to achieve high accuracy.

Exponent
Since the exponent can be positive or negative, some method must be chosen to represent its sign. Two common methods of representing signed integers are sign/magnitude and two's complement. The IEEE binary standard does not use either of these methods to represent the exponent, but instead uses a biased representation for efficient calculation of exponents. In the case of single precision, where the exponent is stored in 8 bits, the bias is 127 (for double precision it is 1023). What this means is that if $\overline{k}$ is the value of the exponent bits interpreted as an unsigned integer (biased), then the exponent of the floating-point number is $\overline{k} - 127$ (unbiased).

	Single	Double
Exponent width in bits	8	11
Format width in bits	32	64
bias	127	1023

For example, in single precision, if the binary integer stored in the exponent field is 00000001, then the exponent of the floating-point number is $-126 = e_{\min}$. Similarly, the exponent $e_{\max} = 127$ is stored in the exponent field by 11111110. What do 00000000 and 11111111 represent? It will be discussed later.

Example Analogously, in our small system, we use bias 7. For example, the binary format 1 0001 100 represents the binary floating-point number -1.100×2^{-6}.

1.3.1 Special quantities

The IEEE standard specifies the following special values: $\pm\infty$, NaNs, ± 0, and denormalized numbers. These special values are all encoded with exponents of either $e_{\max}+1$ or $e_{\min}-1$.

Infinities

There are two infinities: $\pm\infty$. There are two cases where an infinity is produced. When a normal nonzero floating-point number is divided by zero, an infinity is produced. Another case is when overflow occurs. Infinities provide a way to continue the computation in these two cases. The infinity symbol obeys the usual mathematical conventions regarding infinity, such as $\infty + \infty = \infty$, $(-1) * \infty = -\infty$, and $(\text{finite})/\infty = 0$.

Example 1.3.1 *Consider two floating-point numbers* $x = 1.000 \times 2^5$ *and* $y = 1.000 \times 2^0$ *in our small system. Suppose that we want to compute*

$$\sin\theta = \frac{y}{\sqrt{x^2+y^2}} \quad \textit{or} \quad \cos\theta = \frac{x}{\sqrt{x^2+y^2}}.$$

With infinities, if we compute $\sin\theta$, x^2 *overflows to* $+\infty$ *and the computation continues and the computed* $\sin\theta = 0$. *Without infities, the computation may be interrupted or* x^2 *is set to the largest number* 1.111×2^7 *and we will get* $\sin\theta = 1.000 \times 2^{-4}$. *As we know, when* $|y|/|x|$ *is small,* $\sin\theta \approx y/x = 1.000 \times 2^{-5}$. *However, if we compute* $\cos\theta$ *using the above formula, with infinities we will get* $\cos\theta = x/+\infty = 0$. *Without infinities, if* x^2 *is set to the largest number in this case, we will get* $\cos\theta = 2$. *Both are very wrong. A better formula is to replace the denominator* $\sqrt{x^2+y^2}$ *with* $m\sqrt{(1+(y/m)^2}$, *where* $m = \max(|x|,|y|) = x$. *Then we will get the computed* $\sin\theta = 1.000 \times 2^{-5}$ *or* $\cos\theta = 1.000 \times 2^0$.

The infinity symbol is represented by a zero fraction and the same exponent field as a NaN ($e_{\max}+1$); the sign bit distinguishes between $\pm\infty$.

NaNs

Traditionally, the computation of $0/0$ or $\sqrt{-1}$ has been treated as an unrecoverable error which causes a computation to halt. However it makes sense for a computation to continue in such a situation. IEEE standard has NaN (Not a Number). Just as the infinities, NaNs provide a way to continue the computation in situations like $0/0$ and $\sqrt{-1}$. Suppose that the subroutine `zero(f,a,b)` finds the zeros of a function f in the interval $[a,b]$. Normally, we would not require the user to input the domain of f. If in finding a zero, a guess falls outside the domain of f and the function is evaluated at the guess, then it makes sense to continue the computation. For example, $f = (x^2 - 1)/(x - 1)$, if the function is evaluated at the guess $x = 1$, the result is NaN, thus $x = 1$ is rejected and the subroutine continues to find a zero.

Operations that produce a NaN

Operation	NaN Produced By
$+$	$\infty + (-\infty)$
$*$	$0 * \infty$
$/$	$0/0$, ∞/∞
REM	x REM 0, ∞ REM y
sqrt	sqrt(x) when $x < 0$

In IEEE 754, NaNs are represented as floating-point numbers with exponent $e_{\max} + 1$ and nonzero fractions. Thus there is not a unique NaN, but rather a whole family of NaNs. In general, whenever a NaN participates in a floating-point operation, the result is another NaN.

Signed zeros

Zero is represented by the exponent $e_{\min} - 1$ and a zero fraction. Since the sign bit can take on two different values, there are two zeros: ± 0. Naturally, when comparing $+0$ and -0, $+0 = -0$.

Signed zeros can be useful. When $x > 0$ underflows to $+0$, $1/x$ gives $+\infty$; when $x < 0$ underflows to -0, $1/x$ gives $-\infty$. They work like $\lim_{x \to 0^+} 1/x = +\infty$ and $\lim_{x \to 0^-} 1/x = -\infty$. Also, with signed zeros, we have $x = 1/(1/x)$ for $x = \pm\infty$. Without signed zeros, it would cause confusion. Another example involves $\log x$. With signed zeros, we preserve the mathematical indentities $\lim_{x \to 0^+} \log x = -\infty$ and $\lim_{x \to 0^-} \log x = \text{NaN}$.

The above infinities, NaNs, and signed zeros can be similarly introduced into our small floating-point number system. For example, 1 1111 000 represents $-\infty$ and 1 1111 010 represents NaN.

Denormalized numbers

In our small system, $1.000 \times 2^{e_{\min}}$ is stored in the bit format:

0 0001 000

The distance between this number and the next larger number $1.001 \times 2^{e_{\min}}$ is $2^{-t+1} \times 2^{e_{\min}}$. However, its distance to zero is $2^{e_{\min}}$. It makes sense to fill in

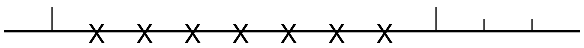


Figure 1.3.1: Denormalized numbers (between 0 and 2^{-6} marked by $\times$).

Exponent	Fraction	Represents
$e = e_{\text{min}} - 1$	$f = 0$	± 0
$e = e_{\text{min}} - 1$	$f \neq 0$	$0.f \times 2^{e_{\text{min}}}$
$e_{\text{min}} \leq e \leq e_{\text{max}}$	$-$	$1.f \times 2^{e}$
$e = e_{\text{max}} + 1$	$f = 0$	∞
$e = e_{\text{max}} + 1$	$f \neq 0$	NaN

Table 1.1: IEEE 754 values

this gap with the numbers $0.001 \times 2^{e_{\text{min}}}, ..., 0.111 \times 2^{e_{\text{min}}}$. These numbers are called denormalized numbers. Figure 1.3.1 plots the positive denormalized numbers in our small floating-point number system.

As shown above, without denormals, the spacing between two floating-point numbers around $\beta^{e_{\text{min}}}$ changes from $\beta^{-t+1}\beta^{e_{\text{min}}}$ to $\beta^{e_{\text{min}}}$. With denormals, adjacent spacing are either the same length or differ by a factor of β. Thus, small numbers gradually underflow to zero.

Example 1.3.2 *In our small floating-point system, if normalized numbers* $x = 1.001 \times 2^{-6}$ *and* $y = 1.000 \times 2^{-6}$*, then* $x - y$ *is too small to be represented in normalized number range and must be flushed to zero (underflow), although* $x \neq y$*. When the denormalized numbers are introduced,* $x - y$ *does not underflow, instead* $x - y = 0.001 \times 2^{-6}$*. The use of denormalized numbers guarantees*

$$x - y = 0 \Longleftrightarrow x = y.$$

In the IEEE standard, the denormals are encoded by $e_{\text{min}} - 1$ in the exponent and a nonzero in the fraction. In this case, there is no hidden bit. For example, using our small system, the bit format

0 0000 001

represents the denormalized number $0.001 \times 2^{e_{\text{min}}}$.

Table 1.1 summarizes IEEE values.

Chapter 2

Floating-point Arithmetic

2.1 Overflow and Underflow

Overflow means that the exponent is too large to be represented in the exponent field. Underflow means that the exponent is too small to be represented in the exponent field. In our small floating-point system, the largest normal number $N_{\max}$ is 1.111×2^7, which is called the overflow threshold. When the result is larger than $N_{\max}$ overflow occurs. The smallest positive normal number $N_{\min}$ is 1.000×2^{-6}, which is called the underflow threshold. With the denormalized numbers, the smallest positive number is $\beta^{-t+1}\beta^{e_{\min}} = 0.001 \times 2^{-6}$. When the result is smaller than $N_{\min}$ underflow occurs.

2.1.1 Avoiding unnecessary over/underflow

Scaling is a commonly used technique to avoid unnecessary underflow and overflow.

The following shows two algorithms for computing the 2-norm of a vector $x = [x_i]$, $i = 1, ..., n$.

Without scaling

```
ssq = 0.0;
for i=1 to n
   ssq = ssq + x(i)*x(i);
end
nrm2 = sqrt(ssq);
```

With scaling

```
scale = 0.0;
ssq = 1.0;
for i=1 to n
   if (x(i)<>0.0)
     if (scale<abs(x(i))
       tmp = scale/x(i);
       ssq = 1.0 + ssq*tmp*tmp;
       scale = abs(x(i));
     else
       tmp = x(i)/scale;
       ssq = ssq + tmp*tmp;
     end
   end
end
nrm2 = scale*sqrt(ssq);
```

2.2 Correctly Rounded Operations

When we apply a floating-point operation to floating-point numbers, the exact result may not fit the format of the floating-point system. We must round the exact result to fit the format of our system.

The IEEE standard requires that the following floating-point operations are correctly rounded:

- arithmetic operations $+$, $-$, $*$, and $/$
- square root and remainder
- conversions of formats

The format conversions include:

- between floating-point formats, e.g., from single to double.
- between floating-point and integer formats.
- from floating-point to integer value (not format).
- between binary and decimal.

Correctly rounded means that result must be the same as if it were computed exactly and then rounded, usually to the nearest floating-point number. For example, if $\oplus$ denotes the floating-point addition, then given two floating-point numbers a and b,

$$a \oplus b = \mathrm{fl}(a + b).$$

For example, suppose that in our small floating-point system $a = 1.110 \times 2^0$ and $b = 1.111 \times 2^{-1}$, then $a+b = 1.01011 \times 2^1$, which does not fit the format of our system. We require that $a \oplus b = \mathrm{fl}(a + b) = 1.011 \times 2^1$.

From the above requirement, $a \oplus b = b \oplus a = \mathrm{fl}(a+b)$. However, $(a \oplus b) \oplus c$ is not necessarily equal to $a \oplus (b \oplus c)$ and $a \otimes (b \oplus c)$ is not necessarily equal to $a \otimes b \oplus a \otimes c$.

Note that the correctly rounded operations do not include transcendental functions. Thus, for example, the following two expressions for x^3:

```
x**3                    x*x*x
```

the right expression is safer than the left.

2.3 Extra Precision

Most processors, including Intel and Motorola, support extra precision, called logn double by the IEEE standard. On those processors, intermediate results are accumulated in 10 byte registers (15 bit exponent and 64 bit significant, no hidden bit), before the final result is stored in 53 bit significant (one hidden bit) in memory. Extra precision produces more accurate results.

One application of extra precision is in solving linear systems $Ax = b$. Suppose that the accuracy of the computed solution x is unsatisfactory. A technique called iterative refinement can be used to improve the accuracy. Here is the algorithm:

1. Compute the residual $r = Ax - b$;
2. Solve for the correction d in $Ad = r$;
3. Update the solution $x = x - d$.

The above procedure is iterated until the correction is sufficiently small. Iterative refinement improves the solution only when the residual is calculated in high precision.

How can you tell whether the system, including the processor and the compiler, on your machine supports extra precision? Using our small number system as an example to illustrate the idea, you can write the expression:

```
x = (1.0 + 1.25e-1)*(1.0 - 1.25e-1) - 1.0;
```

If the result $x = -2^{-6} = -0.015625$, then the system supports extra precision. If $x = 0.0$, then the system does not support extra precision. Why? If the system supports extra precision, say it accumulates intermediate results in 6 bit registers, then it accumulates the intermediate result $(1+2^{-3})(1-2^{-3}) = 1-2^{-6}$ in a 6 bit register in full accuracy. After subracting one from it, we get -2^{-6}. Without extra precision, the intermediate result $1-2^{-6}$ would be rounded to 4 bit significant resulting 1.0, thus the final result is 0.

2.4 Rounding Modes

In the IEEE standard, rounding occurs whenever an operation has a result that is not exact. By default, rounding means round toward nearest. The standard requires that three other rounding modesbe provided, namely round toward 0, round toward $+\infty$, and round toward $-\infty$. The four rounding modes are:

1. Round to $+\infty$: always round up to the first representable floating point number.
2. Round to $-\infty$: always round down to the first representable floating point number.
3. Round to 0: always truncate the digits that after the last representable digit.
4. Round to nearest even: always round to the nearest floating point number. In the case of a tie, the one with its least significant bit equal to zero is chosen.

Example Consider two floating point numbers: $a = b = 1.010 \times 2^1$ in our small floating-point number system. The exact value of $a * b = 1.1001 \times 2^2$. This value is between the two floating-point numbers 1.100×2^2 and 1.101×2^2. Table 2.1 lists the rounded values under different rounding modes.

Why dynamic directed rounding modes? Testing numerical sensitivity. How? Different rounding introduces slightly different rounding errors. If slightly perturbed intermediate results cause significant changes in the final results, the program is sensitive to the perturbation.

Another application of dynamic directed rounding modes is interval arithmetic.

Round	$\to +\infty$	$\to -\infty$	$\to 0$	$\to$ nearest
1.1001×2^2	1.101×2^2	1.100×2^2	1.100×2^2	1.100×2^2

Table 2.1: Four rounding modes

Chapter 3

Error Analysis

3.1 Sources of Errors

When an infinite series is approximated by a finite sum, *truncation error* is introduced. For example, if we use

$$1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots + \frac{x^n}{n!}$$

to approximate

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots + \frac{x^n}{n!} + \cdots,$$

then the truncation error is

$$\frac{x^{n+1}}{(n+1)!} + \frac{x^{n+2}}{(n+2)!} + \cdots.$$

When a continuous problem is approximated by a discrete one, *discretization error* is introduced. For example, from the expansion

$$f(x+h) = f(x) + hf'(x) + \frac{h^2}{2!} f''(\xi), \quad \text{for some} \quad \xi \in [x, x+h],$$

we can use the following approximation:

$$y_h(x) = \frac{f(x+h) - f(x)}{h} \approx f'(x).$$

The discretization error is $E_{\text{dis}} = |f''(\xi)|h/2$.

h	$y_h(1)$	error
10^{-1}	2.85884195487388	$1.40560126415e-1$
10^{-2}	2.73191865578708	$1.36368273280e-2$
10^{-3}	2.71964142253278	$1.35959407373e-3$
10^{-4}	2.71841774707848	$1.35918619434e-4$
10^{-5}	2.71829541991231	$1.35914532646e-5$
10^{-6}	2.71828318698653	$1.35852748429e-6$
10^{-7}	2.71828196396484	$1.35505794585e-7$
10^{-8}	2.71828177744737	$-5.10116753283e-8$
10^{-9}	2.71828159981169	$-2.28647355716e-7$
10^{-10}	2.71827893527643	$-2.89318261570e-6$
10^{-11}	2.71827005349223	$-1.17749668154e-5$
10^{-12}	2.71827005349223	$-1.17749668154e-5$
10^{-13}	2.71338507218388	$-4.89675627517e-3$
10^{-14}	2.66453525910038	$-5.37465693587e-2$
10^{-15}	2.66453525910038	$-5.37465693587e-2$

Table 3.1: Values of $y_h(1)$ and errors using various sizes of h

Note that both truncation error and discretization error have nothing to do with computation. If the arithmetic is perfect (no rounding errors), the discretization error decreases as h decreases. In practice, however, rounding errors are unavoidable. Consider the above example and let $f(x) = e^x$. We computed $y_h(1)$ on a SUN Sparc V in MATLAB 5.20.

Table 3.1 shows that as h decreases the error first decreases and then increases. This is because of the combination of discretization error and rounding errors. In this example, the discretization error is

$$E_{\text{dis}} = \frac{h}{2}|f''(\xi)| \leq \frac{h}{2}e^{1+h} \approx \frac{h}{2}e \quad \text{for small} \quad h.$$

Now, we consider the rounding errors. Let the computed $y_h(x)$ be

$$\begin{aligned} \hat{y}_h(x) &= \frac{(e^{(x+h)(1+\delta_0)}(1+\delta_1) - e^x(1+\delta_2))(1+\delta_3)}{h}(1+\delta_4) \\ &\approx \frac{e^{x+h}(1+\delta_0+\delta_1+\delta_3+\delta_4) - e^x(1+\delta_2+\delta_3+\delta_4)}{h}, \end{aligned}$$

for $|\delta_i| \leq u$ $(i = 0, 1, 2, 3, 4)$. In the above derivation, we assume that δ_i are small so that we ignore the terms like $\delta_i\delta_j$ or higher order. We also assume

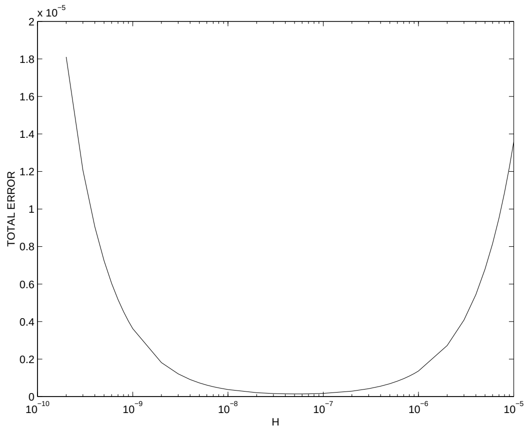


Figure 3.1.1: Total Error

that e^x is computed accurately, that is, the computed e^x equals $e^x(1+\delta)$ where $|\delta| \le u$. Thus we have the rounding error

$$E_{\text{round}} = |y_h(x) - \hat{y}_h(x)| \approx \left| \frac{\xi_1 e^{x+h} - \xi_2 e^x}{h} \right| \quad \text{for} \quad |\xi_1| \le 4u \text{ and } |\xi_2| \le 3u.$$

When $x = 1$, we have

$$E_{\text{round}} \approx \frac{7u}{h} e.$$

So the rounding error increases as h decreases. Combining both errors, we get the total error:

$$E_{\text{total}} = E_{\text{dis}} + E_{\text{round}} \approx \left(\frac{h}{2} + \frac{7u}{h} \right) e.$$

Figure 3.1.1 plots E_{total}.

To minimize the total error, we differentiate E_{total} with respect to h and set the derivative to zero and get the optimal h:

$$h_{\text{opt}} = \sqrt{14u} \approx \sqrt{u}.$$

3.2 Forward and Backward Errors

Suppose a program takes an input x and computes y, we can view the output y as a function of the input x, $y = f(x)$. Denote the computed result as $\hat{y}$, then the absolute error $|y - \hat{y}|$ and the relative error $|y - \hat{y}|/|y|$ are called *forward errors*. Alternatively, we can ask: "For what set of data have we solved our problem?". That is, the computed result $\hat{y}$ is the exact result for the input $x + \Delta x$, i.e., $\hat{y} = f(x + \Delta x)$. In general, there may be many such Δx, so we are interested in minimal such Δx and a bound for $|\Delta x|$. This bound, possibly divided by $|x|$, is called *backward error*.

For example, consider the problem of computing the square root $y = \sqrt{x}$. The IEEE standard requires that the computed square root

$$\hat{y} = \mathrm{fl}(\sqrt{x}) = \sqrt{x}\,(1 + \delta), \quad |\delta| \le u.$$

Then the relative error or the forward error is $|\delta|$, which is bounded by u. What is the backward error? Set

$$\hat{y} = \sqrt{x}\,(1 + \delta) = \sqrt{x + \Delta x},$$

then $\Delta x = 2x\delta + x\delta^2$. Thus, ignoring δ^2, we have the backward error:

$$|\Delta x|/|x| \approx 2|\delta| \le 2u.$$

Example 3.2.1 *Computing the sum:*

$$s_n = x_1 \oplus x_2 \oplus \cdots \oplus x_n$$

Denote the partial sum $s_i = x_1 \oplus x_2 \oplus \cdots \oplus x_i$, *then*

$$\begin{aligned}
s_2 &= x_1 \oplus x_2 = x_1(1 + \epsilon_1) + x_2(1 + \epsilon_1) \\
s_3 &= s_2 \oplus x_3 = (s_2 + x_3)(1 + \epsilon_2) \\
&= x_1(1 + \epsilon_1)(1 + \epsilon_2) + x_2(1 + \epsilon_1)(1 + \epsilon_2) \\
&\quad + x_3(1 + \epsilon_2) \\
s_n &= x_1(1 + \epsilon_1)(1 + \epsilon_2) \cdots (1 + \epsilon_{n-1}) \\
&\quad + x_2(1 + \epsilon_1)(1 + \epsilon_2) \cdots (1 + \epsilon_{n-1}) \\
&\quad + x_3(1 + \epsilon_2) \cdots (1 + \epsilon_{n-1}) \\
&\quad + \cdots \\
&\quad + x_{n-1}(1 + \epsilon_{n-2})(1 + \epsilon_{n-1}) \\
&\quad + x_n(1 + \epsilon_{n-1})
\end{aligned}$$

Define

$$1 + \eta_1 = (1 + \epsilon_1)(1 + \epsilon_2) \cdots (1 + \epsilon_{n-1})$$

$$
\begin{aligned}
1+\eta_2 &= (1+\epsilon_1)(1+\epsilon_2)\cdots(1+\epsilon_{n-1}) \\
1+\eta_3 &= (1+\epsilon_2)\cdots(1+\epsilon_{n-1}) \\
&\cdots \\
1+\eta_{n-1} &= (1+\epsilon_{n-2})(1+\epsilon_{n-1}) \\
1+\eta_n &= (1+\epsilon_{n-1})
\end{aligned}
$$

$$
\begin{aligned}
|\eta_n| &= |\epsilon_{n-1}| \le u \\
1+\eta_{n-1} &= 1+(\epsilon_{n-2}+\epsilon_{n-1})+\epsilon_{n-2}\epsilon_{n-1} \\
|\eta_{n-1}| &\approx |\epsilon_{n-2}+\epsilon_{n-1}| \le 2u
\end{aligned}
$$

In general,

$$
\begin{aligned}
|\eta_1| &\le (n-1)u \\
|\eta_i| &\le (n-i+1)u, \quad i=2,3,...,n
\end{aligned}
$$

A more rigorous bound: If $nu \le 0.1$ *and* $|\epsilon_i| \le u$ $(i=1,2,...,n)$*, then*

$$(1+\epsilon_1)(1+\epsilon_2)\cdots(1+\epsilon_n) = 1+\eta$$

where

$$|\eta| \le 1.06nu.$$

Thus

$$x_1 \oplus \cdots \oplus x_n = x_1(1+\eta_1) + \cdots + x_n(1+\eta_n)$$

$|\eta_1| \le 1.06(n-1)u$
$|\eta_i| \le 1.06(n-i+1)u,\ i=2,3,...,n$

How bad is the accumulation error? Using double precision, $u = 10^{-15}$, for $nu \ge 10^{-4}$, $n \ge 10^{11}$.

There is an accurate and efficient summation formula by Kahan. If $\sum_{i=1}^{n} x_i$ is computed using the following algorithm:

```
s = x(i);             % partial sum
c = 0.0;              % correction
for i=2:n
   y = x(i) - c;      % correction from previous iteration
   t = s + y;         % update partial sum
   c = (t - s) - y;   % update correction
   s = t;
end
```

Then the computed sum s is equal to

$$\sum_{i=1}^{n} x_i(1+\eta_i) + O(nu^2)\sum_{i=1}^{n}|x_i|, \quad \text{where} \quad |\eta_i| \le 2u.$$

A formal proof, much longer and trickier than the algorithm, can be found in [2, Page 615] or [1].

The process of bounding the backward error is called backward error analysis. The motivation is to interpret rounding errors as perturbations in the data. Consequently, it reduces the question of estimating the forward error to perturbation theory. We will see its significance in the following sections.

To illustrate the forward and backward errors, let us consider the computation of $\hat{x} - \hat{y}$, where $\hat{x}$ and $\hat{y}$ can be previously computed results. Assume that x and y are exact results and $\hat{x} = x(1+\delta_x)$ and $\hat{y} = y(1+\delta_y)$, then

$$\hat{x} \ominus \hat{y} = \mathrm{fl}(\hat{x} - \hat{y}) = (\hat{x} - \hat{y})(1+\delta), \quad |\delta| \le u.$$

It then follows that $\hat{x} \ominus \hat{y} = x(1+\delta_x)(1+\delta) - y(1+\delta_y)(1+\delta)$. Ignoring the second order terms $\delta_x\delta$ and $\delta_y\delta$ and letting $\delta_1 = \delta_x + \delta$ and $\delta_2 = \delta_y + \delta$, we get

$$\hat{x} \ominus \hat{y} = x(1+\delta_1) - y(1+\delta_2).$$

If $|\delta_x|$ and $|\delta_y|$ are small, then $|\delta_1|$, $|\delta_2| \le u + \max(|\delta_x|, |\delta_y|)$ are also small i.e., the backward errors are small. However, the forward error (relative error)

$$E_{\mathrm{rel}} = \frac{|(\hat{x} \ominus \hat{y}) - (x-y)|}{|x-y|} = \frac{|x\delta_1 - y\delta_2|}{|x-y|}.$$

If $\delta_1 \ne \delta_2$ i.e., $\delta_x \ne \delta_y$, it is possible that E_{rel} is large when $|x-y|$ is small, i.e., x and y are close to each other. This is called catastrophic cancellation. If $\delta_x = \delta_y$, in particular, if both x and y are original data ($\delta_x = \delta_y = 0$), then $E_{\mathrm{rel}} = \delta$. This is called benign cancellation. The following example illustrates the difference between the two types of cancellations.

Example 3.2.2 *Two algorithms for computing $z = x^2 - y^2$.*

$x \otimes x \ominus y \otimes y$	$(x \oplus y) \otimes (x \ominus y)$
$= (x^2(1+\delta_1) - y^2(1+\delta_2))(1+\delta_3)$	$= (x+y)(1+\delta_1)(x-y)(1+\delta_2)(1+\delta_3)$
$\approx (x(1+\frac{\delta_1+\delta_3}{2}))^2 - (y(1+\frac{\delta_2+\delta_3}{2}))^2$	$= (x^2-y^2)(1+\delta_1+\delta_2+\delta_3)$
$= (x(1+\delta_x))^2 - (y(1+\delta_y))^2$	$= (x(1+\delta_x))^2 - (y(1+\delta_y))^2$
$\lvert\delta_i\rvert \le u$, $\lvert\delta_x\rvert, \lvert\delta_y\rvert \le u$	$\lvert\delta_i\rvert \le u$, $\lvert\delta_x\rvert = \lvert\delta_y\rvert \le 3u/2$

The above backward analysis shows that both algorithms have small backward errors. However, the following forward error analysis shows that the algorithm on the left can produce large error when $|x|$ and $|y|$ are close due to catastrophic cancellation; the algorithm on the right guarantees accurate result by avoiding catastrophic cancellation.

$$\begin{array}{ll} \quad x \otimes x \ominus y \otimes y & \quad (x \oplus y) \otimes (x \ominus y) \\ = (x^2 - y^2)(1+\delta_3) + x^2\delta_1 - y^2\delta_2 & = (x^2 - y^2)(1+\delta_1+\delta_2+\delta_3) \end{array}$$

$$\begin{array}{ll} \quad |\hat{z} - z|/|z| & \quad |\hat{z} - z|/|z| \\ \le |\delta_3| + |x^2\delta_1 - y^2\delta_2|/|z| & = |\delta_1+\delta_2+\delta_3| \\ \le u(1 + (x^2+y^2)/|z|) & \le 3u. \end{array}$$

3.3 Stability of an Algorithm

A method for computing $y = f(x)$ is called backward stable if, for any x, it produces a computed $\hat{y}$ with a small backward error, that is, $\hat{y} = f(x + \Delta x)$ for some small Δx. Usually there exist many such Δx. We are interested in the smallest. If Δx turns out to be large, then the algorithm is unstable.

Example 3.3.1 *Suppose $\beta = 10$ and $t = 3$. Consider the following system:*

$$Ax = b, \quad \textit{where} \quad A = \begin{bmatrix} .001 & 1.00 \\ 1.00 & .200 \end{bmatrix} \textit{ and } b = \begin{bmatrix} 1.00 \\ -3.00 \end{bmatrix}.$$

Applying Gaussian elimination (without pivoting), we get the computed decomposition

$$\widehat{L}\widehat{U} = \begin{bmatrix} 1.00 & 0 \\ 1000 & 1.00 \end{bmatrix} \begin{bmatrix} .001 & 1.00 \\ 0 & -1000 \end{bmatrix}.$$

Let the computed solution

$$\hat{x} = \begin{bmatrix} 0 \\ 1.00 \end{bmatrix}$$

be the exact solution of the perturbed system

$$(A + \Delta A)\hat{x} = b.$$

Solving for ΔA, we get

$$\Delta A = \begin{bmatrix} \times & 0 \\ \times & -3.2 \end{bmatrix}.$$

The smallest ΔA is

$$\Delta A = \begin{bmatrix} 0 & 0 \\ 0 & -3.2 \end{bmatrix},$$

which is of the same size as A. This means that Gaussian elimination (without pivoting) is unstable. Note that the exact solution is

$$x = \begin{bmatrix} -3.2\cdots \\ 1.0032\cdots \end{bmatrix}.$$

3.4 Sensitivity of a Problem

Backward analysis processes the rounding error so that the computed result becomes the exact solution of a perturbed problem. How sensitive is the solution to the perturbations in the data of a problem?

Let us first revisit the problem of evaluating $x^2 - y^2$. Suppose that the data x and y are perturbed by relative errors ϵ_x and ϵ_y respectively. We try to find out how much the relative errors in data can be magnified in the relative error of the result:

$$\begin{aligned} & \frac{|(x(1+\epsilon_x))^2 - (y(1+\epsilon_y))^2 - (x^2 - y^2)|}{|x^2 - y^2|} \\ \approx \; & \frac{|2x^2\epsilon_x - 2y^2\epsilon_y|}{|x^2 - y^2|} \\ \leq \; & \frac{2|x^2 + y^2|}{|x^2 - y^2|}\,\epsilon, \quad \epsilon = \max(|\epsilon_x|, |\epsilon_y|). \end{aligned}$$

The magnification $2|x^2 + y^2|/|x^2 - y^2|$ is called the *condition number* for the problem. When $|x|$ is close to $|y|$ the condition number is large, that is, the result $x^2 - y^2$ can be very sensitive to the perturbations in x and y.

Note that the condition number is an upper bound for any method. Example 3.2.2 shows that the algorithm on the left can achieve the upper bound while the well designed algorithm on the right produces accurate result even when the condition number is large, that is, $|x|$ is close to $|y|$.

Then we revisit the problem of solving a system of linear equations:

$$Ax = b$$

where A and b are known variables and x is the result. The question is: How sensitive is x to the change in A and/or b?. We can assume that the change is only in b since the change in A can be transformed into the change in b. Let $\breve{x}$ be the solution of the perturbed system:

$$A\breve{x} = b + \Delta b.$$

The change in x (relative error) is $\|\breve{x} - x\|/\|x\|$ and the change in b is $\|\Delta b\|/\|b\|$. We use the ratio of the two errors as the condition number:

$$\text{cond} = \frac{\|\breve{x} - x\|/\|x\|}{\|\Delta b\|/\|b\|} = \frac{\|A^{-1}\Delta b\|}{\|x\|} \cdot \frac{\|Ax\|}{\|\Delta b\|} \leq \|A^{-1}\| \, \|A\|.$$

So $\|A^{-1}\| \, \|A\|$ is the condition number of the problem of solving a linear system.

Example 3.4.1 *A metal bar with one end fixed on a wall and another end carrying a mass. To find out how the bar bends, we need to solve a symmetric pentadiagonal linear system whose coefficient matrix has the form when $N = 8$:*

$$\begin{bmatrix} 9 & -4 & 1 & & & & & \\ -4 & 6 & -4 & 1 & & & & \\ 1 & -4 & 6 & -4 & 1 & & & \\ & 1 & -4 & 6 & -4 & 1 & & \\ & & 1 & -4 & 6 & -4 & 1 & \\ & & & 1 & -4 & 6 & -4 & 1 \\ & & & & 1 & -4 & 5 & -2 \\ & & & & & 1 & -2 & 1 \end{bmatrix}.$$

The condition number of the matrix grows in the order of N^4 as shown in the following table.

N	10	100	1000	10000
cond	1.4e+04	1.3e+08	1.3e+12	1.3e+16

To reduce the discretization error, we need a large N, usually in thousands. However, when N is large, the system is ill-conditioned, thus the computed solution is severely contaminated by rounding errors. The iterative refinement can attenuate the risk.

In general, we can view a problem with data x and result y as a function $y = f(x)$. The result of the perturbed problem is $\breve{y} = f(x + \Delta x)$. The sensitivity is measured by

$$\text{cond} = \frac{|\breve{y} - y|/|y|}{|\Delta x|/|x|} = \frac{|f(x + \Delta x) - f(x)|}{|\Delta x|} \frac{|x|}{|f(x)|} \approx |f'(x)| \frac{|x|}{|f(x)|}. \quad (3.4.1)$$

Note that the conditioning of a problem is independent of rounding errors and algorithms for solving the problem. The following example is due to Wilkinson(see [3].

Example 3.4.2 *Let* $p(x) = (x-1)(x-2)...(x-19)(x-20) = x^{20} - 210x^{19} + ...$ *The zeros of* $p(x)$ *are* $1, 2, ..., 19, 20$ *and well separated. With the floating-point number system of* $\beta = 2, t = 30$ *we enter a typical coefficient into the computer, it is necessary to round it to* 30 *siginificant base-2 digits. If we make a change in the* $30th$ *significant base-2, only one of the twenty coefficients, the coefficient of* x^{19}*, is changed from* -210 *to* $-210+2^{-23}$*. Let us see how much effect this small change has on the zeros of the polynomial. Here we list (using* $\beta = 2$*,* $t = 90$*) the roots of the equation* $p(x)+2^{-23}x^{19} = 0$*, correctly rounded to the number of digits:*

1.00000 0000	10.09526 6145 $\pm$ 0.64350 0904i
2.00000 0000	11.79363 3881 $\pm$ 1.65232 9728i
3.00000 0000	13.99235 8137 $\pm$ 2.51883 0070i
4.00000 0000	16.73073 7466 $\pm$ 2.81262 4894i
4.99999 9928	19.50243 9400 $\pm$ 1.94033 0347i
6.00000 6944	
6.99969 7234	
8.00726 7603	
8.91725 0249	
20.84690 8101	

Note the small change in the coefficient -210 *has caused ten of the zeros to become complex and that two have moved more than* 2.81 *units off the real axis. That means the zeros of* $p(x)$ *are very sensitive to the change in coefficients. The results were computed under a very accurate computation. They did not get any side effects from rounding errors, and nor is it a problem that the algorithm used solve this problem make some ill-effects on the results. Actually the problem is the matter of sensitivity itself.*

As discussed before, backward error analysis transforms rounding errors into perturbations of data. Thus we can establish a relation between forward and backward errors and the conditioning of the problem. Clearly, (3.4.1) shows that

$$E_{\text{forward}} \leq \text{cond} \cdot E_{\text{backward}}.$$

This inequality tells us that large forward errors can be caused by either ill-conditioning of the problem or unstable algorithm, or both. The significance of backward error analysis is that it allows us to determine whether an algorithm is stable (small backward errors). If we can prove the algorithm is stable, in other words, the backward errors are small, say, no larger than the measurement errors in data, then we know that large forward errors are due to the ill-conditioning of the problem. On the other hand, if we know the problem is well-conditioned, then large forward errors must be caused by unstable algorithm.

As discussed in Section 2.4, dynamic directed rounding can be used to test a program's sensitivity to perturbations.

Bibliography

[1] David Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 23(1):5–48, 1991.

[2] Donald E. Knuth. *The Art of Computer Programming*, volume II. Addison-Wesley, Reading Mass,, 2 edition, 1997.

[3] J.H. Wilkinson. *Rounding Errors in Algebraic Processes.* Prentice Hall, Englewood Cliffs, N.J., 1963.

Index