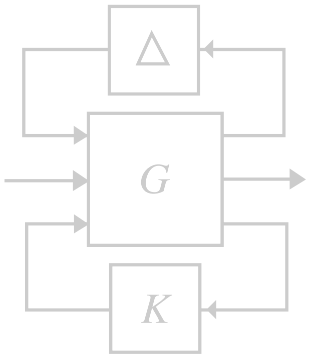

Chapter 2
Floating Point Arithmetic

2.1 Introduction

2.2 Floating Point Numbers

Theorem 2.1. *Let $\beta \geq 2$ be an integer. For every $x \in I\!R$ there exist integers e and $d_i \in \{0, \dots, \beta - 1\}$, $i = 0, 1, \dots,$ such that*

$$x = sign(x) \left(\sum_{i=0}^{\infty} d_i \beta^{-i} \right) \beta^e. \tag{2.1}$$

The representation (2.1) *is unique if one requires that $d_0 > 0$.*

In a computer only a subset of all real numbers can be represented. These are the so–called *floating point numbers* and they are of the form

$$\bar{x} = (-1)^s \left(\sum_{i=0}^{t-1} d_i \beta^{-i} \right) \beta^e, \tag{2.2a}$$

$$\text{with } d_i \in \{0, \dots, \beta - 1\}, i = 0, 1, \dots, t-1, \quad \text{and } e \in \{e_{\min}, \dots, e_{\max}\}.$$

The integer β is called the *base*, $\sum_{i=0}^{t-1} d_i \beta^{-i}$ is the *significant* or *mantissa*, t is the *mantissa length*, e is the *exponent*, and $\{e_{\min}, \dots, e_{\max}\}$ is the *exponent range.*

A *floating point number system* is the set of all numbers that are of the form (2.2). A floating point number system can be characterized by the four integers

$$\beta, t, e_{\min}, e_{\max}.$$

If $\beta = 2$, then we say the floating point number system is a *binary system.* In this case the d_i's are called *bits.* If $\beta = 10$, then we say the floating point number system is a *decimal system.* In this case the d_i's are called *digits.*

In this chapter we use $\bar{x}$ to denote a floating point number.

A *floating point number is called normalized* if

$$d_0 = 0 \Leftrightarrow x = 0.$$

Example 2.2 A sketch of the floating point number system ($\beta = 2, t = 3, e_{\min} = -1, e_{\max} = 2$) is given in Figure 2.1. Notice that the normalized floating point numbers $\bar{x} \neq 0$ are of the form

$$\bar{x} = 1.d_1 d_2 \times 2^e$$

since the normalization condition implies that $d_0 \in \{1, \dots, \beta - 1\} = \{1\}$.

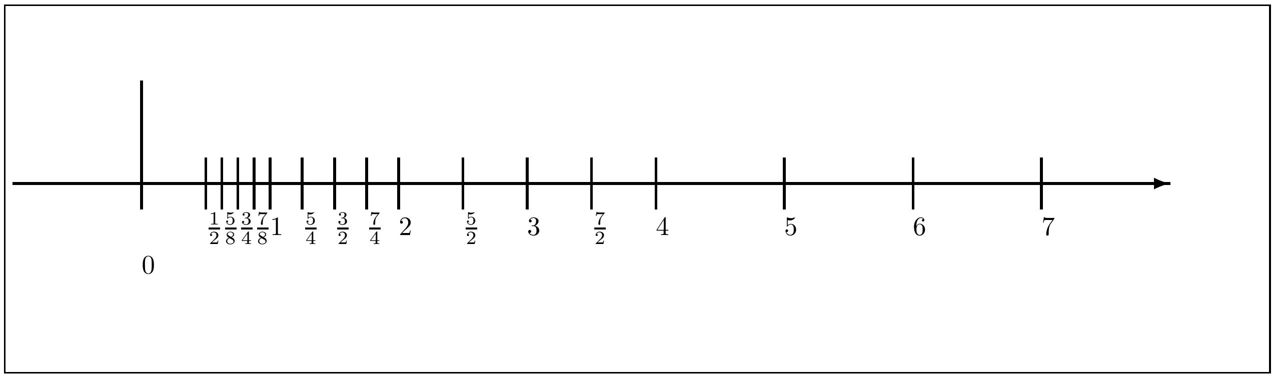

Figure 2.1: The floating point number system ($\beta = 2, t = 3, e_{\min} = -1, e_{\max} = 2$). Only positive numbers are shown.

◇

Let us consider a floating point number system which is characterized by $(\beta, t, e_{\min}, e_{\max})$. A few properties of this floating point number system are summarized below.

- The mantissa satisfies

$$\sum_{i=0}^{t-1} d_i \beta^{-i} \leq \sum_{i=0}^{t-1} (\beta - 1)\beta^{-i} = \beta(1 - \beta^{-t}) < \beta. \tag{2.3}$$

- The mantissa of a normalized floating point number is always greater or equal to one.
- The largest floating point number is

$$\bar{x}_{\max} = \left(\sum_{i=0}^{t-1} (\beta - 1)\beta^{-i} \right) \beta^{e_{\max}} = (1 - \beta^{-t})\beta^{e_{\max}+1}.$$

- The smallest positive normalized floating point number is

$$\bar{x}_{\min} = \beta^{e_{\min}}.$$

- The distance between $\bar{x} = 1$ and the next largest floating point number is $\beta^{-(t-1)}$. In fact, the spacing between the floating point numbers in $[1, \beta]$ is $\beta^{-(t-1)}$. The spacing between the floating point numbers in $[\beta^e, \beta\beta^e]$ is $\beta^{-(t-1)}\beta^e$.

Outside the range $[-(1-\beta^{-t})\beta^{e_{\max}+1}, (1-\beta^{-t})\beta^{e_{\max}+1}]$ the number *overflows* and various things could happen; numbers with absolute value in $(0, \beta^{e_{\min}})$ *underflow* and, again, various things could happen.

If the number x neither overflows nor underflows we define

$$\mathsf{fl}(x) = \bar{x} \text{ the normalized floating point number closest to } x. \tag{2.4}$$

The floating point number $\bar{x}$ closest to x can be obtained by rounding. Suppose that

$$x = sign(x) \left(\sum_{i=0}^{\infty} d_i \beta^{-i} \right) \beta^e, \tag{2.5a}$$

then

$$\mathsf{fl}(x) = \begin{cases} sign(x)\left(\sum_{i=0}^{t-1} d_i\beta^{-i}\right)\beta^e, & \text{if } d_t < \frac{1}{2}\beta, \\ sign(x)\left(\sum_{i=0}^{t-1} d_i\beta^{-i} + \beta^{-(t-1)}\right)\beta^e, & \text{if } d_t \geq \frac{1}{2}\beta. \end{cases} \tag{2.5b}$$

Example 2.3 Let $\beta = 10$, $t = 3$. Then

$$\begin{aligned} \mathsf{fl}(1.234 * 10^{-1}) &= 1.23 * 10^{-1}, \\ \mathsf{fl}(1.235 * 10^{-1}) &= 1.24 * 10^{-1}, \\ \mathsf{fl}(1.295 * 10^{-1}) &= 1.30 * 10^{-1}. \end{aligned}$$

$\diamond$

What is the error incurred by rounding?

Theorem 2.4. *If x is a number within the range of floating point numbers and $|x| \in [\beta^{e-1}, \beta^e)$, then the* absolute error *is given by*

$$|\mathsf{fl}(x) - x| \leq \frac{1}{2}\beta^{e-t} \tag{2.6}$$

and, provided $x \neq 0$, the relative error *is given by*

$$\frac{|\mathsf{fl}(x) - x|}{|x|} \leq \frac{1}{2}\beta^{1-t}. \tag{2.7}$$

Proof. If $x = 0$, then $\mathsf{fl}(x) = x$ and (2.6) follows immediately. We now consider the case $x > 0$ and $x \in [\beta^{e-1}, \beta^e)$. In the interval $[\beta^{e-1}, \beta^e]$ the floating point numbers are uniformly spaced with a separation of β^{e-t}. The closest one to x is $\mathsf{fl}(x)$ and must be within a distance of $\frac{1}{2}\beta^{e-t}$ from x. That is,

$$|\mathsf{fl}(x) - x| \leq \frac{1}{2}\beta^{e-t}.$$

Since $\beta^{e-1} \leq x$ we have (2.7).

The case $x < 0$ can be treated analogously. $\square$

The number

$$\mathbf{u} = \frac{1}{2}\beta^{1-t} \tag{2.8}$$

is called *unit roundoff*. Sometimes it is also called *machine precision* and denoted by ϵ_{mach}.

The relation (2.7) implies that

$$\mathsf{fl}(x) = x(1 + \epsilon), \text{ with } |\epsilon| \leq \mathbf{u}. \tag{2.9}$$

Example 2.5 Let $\beta = 10$, $t = 3$, thus $\mathbf{u} = 5 * 10^{-3}$.

$$\begin{aligned} &|\mathsf{fl}(1.234 * 10^{-1}) - 1.234 * 10^{-1}| = 0.0004, \\ &\frac{|\mathsf{fl}(1.234 * 10^{-1}) - 1.234 * 10^{-1}|}{1.234 * 10^{-1}} = \frac{0.0004}{1.234 * 10^{-1}} \approx 3.2 * 10^{-3}, \\ &|\mathsf{fl}(1.295 * 10^{-1}) - 1.295 * 10^{-1}| = 0.0005, \\ &\frac{|\mathsf{fl}(1.295 * 10^{-1}) - 1.295 * 10^{-1}|}{1.295 * 10^{-1}} = \frac{0.0005}{1.295 * 10^{-1}} \approx 3.9 * 10^{-3}. \end{aligned}$$

◇

To estimate the unit roundoff it is useful to note that

$$\mathsf{fl}(1+\epsilon) = 1 \quad \text{for all } \epsilon \in (-\mathbf{u}, \mathbf{u}). \tag{2.10}$$

This follows from (2.5). In fact, if $\epsilon \in [0, \mathbf{u})$, then

$$1 + \epsilon = + \left(1 + 0\beta^{-1} + \ldots + 0\beta^{-(t-1)} + d_t\beta^{-t} + d_{t+1}\beta^{-(t+1)} + \ldots \right) \beta^0,$$

with $d_t < \frac{1}{2}\beta$. Thus, application of (2.5b) gives

$$\mathsf{fl}(1+\epsilon) = 1.$$

The case $\epsilon \in (-\mathbf{u}, 0)$ can be treated analogously. The observation (2.10) can be used to estimate the unit roundoff with the following algorithm.

Algorithm 2.2.1 Estimation of Unit Roundoff $\mathbf{u}$

Output: The algorithm returns s with $s < \mathbf{u} \le 2s$.

```
1   $s = 1$
2   $s1 = 1 + s$
3   While $s1 > 1$ do
4       $s = s/2$
5       $s1 = 1 + s$
6   End
```

In Algorithm 2.2 we use division by 2 to decrease s. This division can be done exactly if the computer uses a floating point system with base $\beta = 2^p$.

2.3 The IEEE Standard

Most computers use the IEEE standard 754 for binary floating point arithmetic, which we sketch below.

IEEE Single Precision

IEEE single precision normalized floating point numbers are of the form

$$\bar{x} = (-1)^s \times 2^{e-127} \times 1.f \ .$$

First note that the requirement $d_0 \in \{1, \ldots, \beta - 1\}$ for a nonzero normalized number implies $d_0 = 1$. Thus the leading bit of a normalized number is implicitly given and the mantissa is of the form $1.f$, where f is the *fraction*. Next, observe that the form $e - 127$ of the exponent yields negative exponents if $e \in \{0, \ldots, 126\}$; we do not need an extra bit for the sign of the exponent.

An IEEE single precision number is stored in 32 bits using the following format:

s	exp[30:23]		fraction[22:0]	
31	30	23	22	0

Here s is the sign bit. The bit $s = 0$ corresponds to positive numbers, $s = 1$ corresponds to negative numbers. The integer e in the exponent of $\bar{x}$ is of the form

$$e = \sum_{i=0}^{7} e_i 2^i, \quad e_i \in \{0, 1\}.$$

However, only exponents e with $0 < e < 255$ are used for the representation of normalized numbers. Thus the exponent range is $e_{\min} = 1 - 127 = -126$ and $e_{\max} = 254 - 127 = 127$. The exponents $e = 0, e = 255$ are reserved for other representations.

The exponent $e = 0$ is used for the representation of *subnormal numbers* which are of the form

$$(-1)^s \times 2^{-126} \times 0.f \ (f \neq 0).$$

and zero ($(-1)^s \times 0.0$). Subnormal numbers are introduced to reduce the gap between zero and the normalized numbers. The presence of subnormal numbers reduces the influence of underflow.

Consider for example the system of normalized floating point numbers that are of the form

$$(-1)^s \times 2^{e-2} \times 1.f$$

with $e \in \{1, 2, 3, 4\}$. The positive normalized numbers are sketched in Figure 2.2 (bold lines). See also Figure 2.1. The subnormal numbers in this system would be of the form

$$(-1)^s \times 2^{-1} \times 0.f \ \ (f \neq 0).$$

These are the numbers $-3/8, -1/4, -1/8, 3/8, 1/4, 1/8$ and are indicated by the thin lines in Figure 2.2.

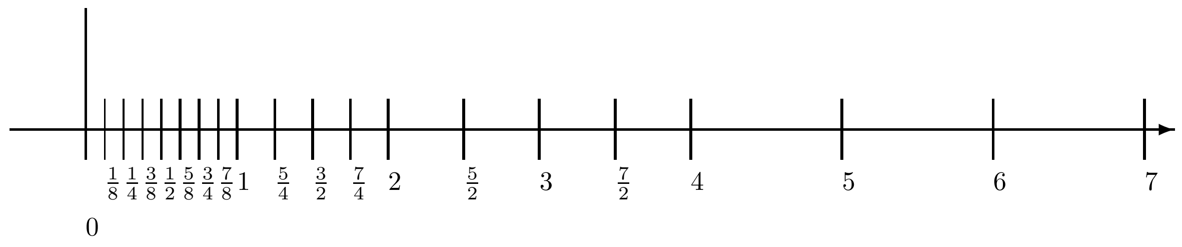
$\frac{1}{8}$ $\frac{1}{4}$ $\frac{3}{8}$ $\frac{1}{2}$ $\frac{5}{8}$ $\frac{3}{4}$ $\frac{7}{8}$ 1 $\frac{5}{4}$ $\frac{3}{2}$ $\frac{7}{4}$ 2 $\frac{5}{2}$ 3 $\frac{7}{2}$ 4 5 6 7
0

Figure 2.2: A Floating Point Number System with Normal and Subnormal Numbers.

There are special quantities that you may have seen in previous computations. These quantities are introduced to make debugging of codes easier. `NaN` (not a number) is assigned to the result of calculations like $0/0$ or $\sqrt{-1}$. `NaN` is represented by $e = 255, f \neq 0$. The quantities $\pm$`INF` occur if one tries to divide a nonzero number by zero. `INF` corresponds to $(e = 255, f = 0)$.

The IEEE Single Precision Storage Format is summarized in Table 2.1.

IEEE Double Precision

IEEE double precision floating point numbers are stored in (32+32=) 64 bits using the following format:

s	exp[52:62]	fraction[51:32]
63	62 52	51 32

fraction[31:0]
31 0

Table 2.1. *IEEE Single Precision Storage Format*

Single Precision Value	IEEE Single Precision Representation
Normalized value ($0 < e < 255$)	$(-1)^s \times 2^{e-127} \times 1.f$
Subnormal value ($e = 0, f \neq 0$)	$(-1)^s \times 2^{-126} \times 0.f$
Signed Zero ($e = 0, f = 0$)	$(-1)^s \times 0.0$
$-$`INF` ($e = 255, f = 0$)	$s = 0; e = 255; f = 0$ (all bits in f are zero)
$+$`INF` ($e = 255, f = 0$)	$s = 1; e = 255; f = 0$ (all bits in f are zero)
`NaN` ($e = 255, f \neq 0$)	

Table 2.2. *IEEE Values*

Common Name	(Approximate) Equivalent Value	
	Single Precision	Double Precision
Unit roundoff	$2^{-24} \approx 6.e-8$	$2^{-53} \approx 1.1e-16$
Maximum normal number	$3.4e+38$	$1.7e+308$
Minimum positive normal number	$1.2e-38$	$2.3e-308$
Maximum subnormal number	$1.1e-38$	$2.2e-308$
Minimum positive subnormal number	$1.5e-45$	$5.0e-324$

The double precision values are summarized in Tables 2.3 and 2.2.

Table 2.3. *IEEE Double Precision Storage Format*

Single Precision Value	IEEE Double Precision Representation
Normalized value ($0 < e < 2047$)	$(-1)^s \times 2^{e-1023} \times 1.f$
Subnormal value ($e = 0, f \neq 0$)	$(-1)^s \times 2^{-1022} \times 0.f$
Signed Zero ($e = 0, f = 0$)	$(-1)^s \times 0.0$
$-$`INF` ($e = 2047, f = 0$)	$s = 0; e = 2047; f = 0$ (all bits in f are zero)
$+$`INF` ($e = 2047, f = 0$)	$s = 1; e = 2047; f = 0$ (all bits in f are zero)
`NaN` ($e = 2047, f \neq 0$)	

IEEE Floating Point Operations

If we can only represent a limited number of numbers, then the question arises with what accuracy numbers can be added, subtracted, multiplied, or divided. If $\Box$ represents one of the elementary operations $+, -, *, /$ and if $\bar{x}$ and $\bar{y}$ are floating point numbers, then $\bar{x}\Box\bar{y}$ is not guaranteed to be a floating point number. Thus, what is the computed value for $\bar{x}\Box\bar{y}$? In IEEE floating point arithmetic the result of the computation $\bar{x}\Box\bar{y}$ is equal to the floating point number that is nearest to the exact result $\bar{x}\Box\bar{y}$. Therefore we use $\mathsf{fl}(\bar{x}\Box\bar{y})$ to denote the result of the computation $\bar{x}\Box\bar{y}$

We can use the following model for the computation of $\bar{x}\Box\bar{y}$, where $\Box$ is one of the elementary operations $+, -, *, /$.

1. Given floating point numbers $\bar{x}$ and $\bar{y}$.
2. Compute $\bar{x}\Box\bar{y}$ exactly.
3. Round the exact result $\bar{x}\Box\bar{y}$ to the nearest floating point number and normalize the result.

The actual implementation of the elementary operations is more sophisticated. For more details see, e.g., [**?**]. Given two numbers $\bar{x}, \bar{y}$ in IEEE floating point format, the IEEE floating point arithmetic guarantees that

$$\frac{|\mathsf{fl}(\bar{x}\Box\bar{y}) - (\bar{x}\Box\bar{y})|}{\bar{x}\Box\bar{y}} \leq \mathbf{u}, \tag{2.11}$$

if no overflow or underflow occurs, c.f. (2.6).

Printing Floating Point Numbers

Internally, IEEE floating point numbers are stored in binary format. However, when printed, the numbers are converted into the decimal system. This conversion introduces an error. This error depends on the format with which the number is printed. Consider the following diary from a Matlab session. Matlab uses IEEE double precision arithmetic. This Matlab session was run on a SUN Ultra 10 workstation using Matlab Version 5.3.0.10183 (R11), Jan 21, 1999.

```
>> x = 1/6;
>> x

x = 0.1667

>> format short e
>> x

x = 1.6667e-01

>> format long e
>> x

x = 1.666666666666667e-01

>> fprintf(1,' x = %30.15e \n', x)
   x =          1.666666666666667e-01

>> fprintf(1,' x = %30.16e \n', x)
   x =         1.6666666666666666e-01
```

```
>> fprintf(1,' x = %30.17e \n', x)
   x =        1.66666666666666657e-01

>> fprintf(1,' x = %30.25e \n', x)
   x = 1.6666666666666665741480813e-01
```

The number $x = 1/6$ can not be represented exactly in a binary floating point system with finite mantissa. Thus, an error occurs when $x = 1/6$ is rounded to the nearest IEEE floating point number. The different outputs of $\mathsf{fl}(x)$ are summarized below.

$$x = 1.6667e-01, \qquad (2.12a)$$
$$x = 1.666666666666667e-01, \qquad (2.12b)$$
$$x = 1.666666666666667e-01, \qquad (2.12c)$$
$$x = 1.6666666666666666e-01, \qquad (2.12d)$$
$$x = 1.6666666666666665741480813e-01. \qquad (2.12e)$$

Using `format long e` or `fprintf(1,' x = %30.15e \n', x)` shows 1/6 correctly rounded to 16 digits, see (2.12b) and (2.12c). Printing $x = 1/6$ using `format short e` shows 1/6 correctly rounded to 5 digits, see (2.12a). However, information is lost because only 5 digits are requested for output. On the other hand, if we request an output with a higher accuracy than supported by the floating point system, then the last digits can not be trusted. This the case when the printing formats `fprintf(1,' x = \%30.16e \n', x)` or `fprintf(1,' x = \%30.25e \n', x)` are used, see (2.12d) and (2.12e).

2.4 The Effects of Rounding Errors

2.4.1 Cancellation

For the estimate (2.11) to hold it is essential that $\bar{x}$ and $\bar{y}$ are both floating point numbers.

Example 2.6 Consider the floating point system $\beta = 10$ and $t = 4$. [1] If we subtract the floating point numbers $\bar{x} = 2.552 * 10^3$ and $\bar{y} = 2.551 * 10^3$, then $\bar{x} - \bar{y} = 0.001 * 10^3 = 1.000 * 10^0$. The result is a floating point number and in this case no error aoccurs in the subtraction of these numbers.

If we subtract the floating point numbers $\bar{x} = 2.552 * 10^3$ and $\bar{y} = 2.551 * 10^2$, then

$$\bar{x} - \bar{y} = 2.2969 * 10^3.$$

This is not a floating point number. Thus, the floating point result of this subtraction is $\mathsf{fl}(\bar{x}-\bar{y}) = 2.297*10^3$. We see that

$$\frac{|\mathsf{fl}(\bar{x}-\bar{y}) - (\bar{x}-\bar{y})|}{|\bar{x}-\bar{y}|} = \frac{|2.297 * 10^3 - 2.2969 * 10^3|}{2.2969 * 10^3} \approx 4.4 * 10^{-5} < \mathbf{u} = 5 * 10^{-5}.$$

This agrees with (2.7). ◇

The addition/subtraction of two floating point numbers $\bar{x}$ and $\bar{y}$ in general only leads to *benign cancellations.*

The cancellation error can be much larger if we add/subtract two numbers that are not in floating point format. Problems may occur if one subtracts two numbers with same sign and of approximately the same size (or if one adds two numbers with opposite sign and with absolute values of approximately the

[1] The effects of canellation are easier to visualize if we use the decimal system. Your computer most likely uses the binary system. Some cancellation effects are harder to recongnize because the binary numbers are converted to decimals for the output. However, cancellation still occurs and the effects are the same.

same size). In this case the most significant digits (left in the mantissa) match and cancel each other. To normalize the result, the result will be shifted to the left and the last digits are filled with (spurious) zeros. This is called *catastrophic cancellation*.

Example 2.7 Consider the floating point system $\beta = 10$ and $t = 4$. If we subtract the numbers $x = 2.5515052 * 10^3$ and $y = 2.5514911 * 10^3$ (notice that x and y are not in floating point format), then in floating point arithmetic the result is obtained as follows:

1. Compute the floating point numbers $\bar{x}$ and $\bar{y}$ nearest to x and y, respectively: $\bar{x} = 2.552 * 10^3$ and $\bar{y} = 2.551 * 10^3$.

2. Compute $\bar{x} - \bar{y}$ exactly: $\bar{x} - \bar{y} = 0.001 * 10^3$.

3. Round the exact result $\bar{x} - \bar{y}$ to the nearest floating point number: $\mathsf{fl}(0.001 * 10^3) = 0.001 * 10^3$. Shift the result to the left to normalize the number: $\mathsf{fl}(0.001 * 10^3) = 1.000$. The last digits are filled with (spurious) zeros.

The exact result is $2.5515052 * 10^3 - 2.5514911 * 10^3 = 1.410 * 10^{-2}$. The relative error between exact and computed solution is

$$\frac{|1.000 - 1.410 * 10^{-2}|}{1.410 * 10^{-2}} \approx 70 \gg \mathbf{u} = 5 * 10^{-4}.$$

Note that this large error is not due the computation of $\mathsf{fl}(\bar{x} - \bar{y})$. In fact this operation could be computed exactly. The large error is caused by the rounding of x and y at the beginning. $\diamond$

It is important to notice that *catastrophic cancellation can not be detected by looking at the numbers.* The zeros that showed up in the previous example after shifting the digits and normalizing the number are only visible in the decimal system. Since most computers use binary systems, these binary zeros will be converted to nonzero digits!

We can give a formal analysis of the error incurred by the subtraction of two numbers:

Suppose that $\bar{x}$ and $\bar{y}$ are the floating point approximations of x and y, respectively. Then

$$\bar{x} = x(1 + \epsilon_1), \quad \bar{y} = y(1 + \epsilon_2), \text{ with } |\epsilon_1|, |\epsilon_2| \le \mathbf{u},$$

see (2.9). Moreover

$$\mathsf{fl}(\bar{x} - \bar{y}) = (\bar{x} - \bar{y})(1 + \epsilon_3), \text{ with } |\epsilon_3| \le \mathbf{u}.$$

Thus,

$$\begin{aligned}\mathsf{fl}(\mathsf{fl}(x) - \mathsf{fl}(y)) = \mathsf{fl}(\bar{x} - \bar{y}) &= (\bar{x} - \bar{y})(1 + \epsilon_3) = [x(1 + \epsilon_1) - y(1 + \epsilon_2)](1 + \epsilon_3) \\ &= (x - y)(1 + \epsilon_3) + (x\epsilon_1 - y\epsilon_2)(1 + \epsilon_3)\end{aligned}$$

and, if $x - y \neq 0$, then the relative error is given by

$$\frac{|\mathsf{fl}(\mathsf{fl}(x) - \mathsf{fl}(y)) - (x - y)|}{|x - y|} = \left|\epsilon_3 + \frac{x\epsilon_1 - y\epsilon_2}{x - y}(1 + \epsilon_3)\right| \tag{2.13}$$

If $\epsilon_1\epsilon_2 \neq 0$ and $x - y$ is small, then the quantity on the right hand side could be big. In this analysis it is not important that $\bar{x}$ and $\bar{y}$ are the floating point approximations of x and y, respectively. We can view $\bar{x}$ and $\bar{y}$ more generally as perturbations of x and y.

Do we have to expect large errors also for other calculations? Similar to the derivation of (2.13) one can show that

$$\frac{|\mathsf{fl}(\mathsf{fl}(x) + \mathsf{fl}(y)) - (x+y)|}{|x+y|} = \big|\epsilon_4 + \frac{x\epsilon_1 + y\epsilon_2}{x+y}(1+\epsilon_4)\big|, \tag{2.14}$$

$$\frac{|\mathsf{fl}(\mathsf{fl}(x) * \mathsf{fl}(y)) - (x*y)|}{|x*y|} = |\epsilon_1 + \epsilon_2 + \epsilon_1\epsilon_2 + (\epsilon_1 + \epsilon_2 + \epsilon_1\epsilon_2)\epsilon_5| \approx |\epsilon_1 + \epsilon_2|, \tag{2.15}$$

$$\frac{|\mathsf{fl}(\mathsf{fl}(x)/\mathsf{fl}(y)) - (x/y)|}{|x/y|} = \big|\frac{1+\epsilon_1}{1+\epsilon_2}(1+\epsilon_6) - 1)\big| \approx \big|\epsilon_1 - \epsilon_2\big|. \tag{2.16}$$

Thus, catastrophic cancellation can only occur if one subtracts two numbers which are not both in floating point format and which have the same sign and are of approximately the same size, see (2.13)*, or if one adds two numbers which are not both in floating point format, which have opposite sign and their absolute values of approximately the same size, see* (2.14)*.*

2.4.2 Examples

Example 2.8 Suppose we want to evaluate $1 - \cos(x)$ near $x = 0$. Since $\cos(0) = 1$ we expect catastrophic cancellation. The computations of $1 - \cos(x)$ at various x near zero are shown in the following table. All computations were done on a SUN SparcStation 10 in single precision Fortran.

x	$1 - \cos$
0.500000	0.122417
0.125000	$0.780231E-02$
$0.312500E-01$	$0.488222E-03$
$0.781250E-02$	$0.305176E-04$
$0.195312E-02$	$0.190735E-05$
$0.488281E-03$	$0.119209E-06$
$0.122070E-03$	0.
$0.305176E-04$	0.
$0.762939E-05$	0.
$0.190735E-05$	0.
$0.476837E-06$	0.
$0.119209E-06$	0.
$0.298023E-07$	0.

In this case we can avoid catastrophic cancellation for evaluations near $x = 0$. For example, since $\cos^2(x) + \sin^2(x) = 1$ it holds that $1 - \cos(x) = \sin^2(x)/(1 + \cos(x))$. Alternatively one may use of the Taylor expansion of $cos(x)$:

$$\cos(x) = 1 - \frac{x^2}{2} + \frac{x^4}{4!} - \frac{x^6}{6!} + \frac{x^8}{8!} \pm \ldots .$$

The Leibnitz criterion[2] guarantees that

$$\Big|\cos(x) - \left(1 - \frac{x^2}{2} + \frac{x^4}{4!} - \frac{x^6}{6!}\right)\Big| < \frac{x^8}{8!}.$$

Hence the truncated Taylor series is a good approximation if x is small. After some rearrangements we can use the approximation

$$1 - \cos(x) \approx \frac{x^2}{2}\left(1 - \frac{x^2}{12} + \frac{x^4}{360}\right)$$

[2]The Leibnitz criterion says that if the series $S = \sum_{i=1}^{\infty}(-1)^i c_i$, $c_i \geq 0$, converges, then $\left|S - \sum_{i=1}^{n}(-1)^i c_i\right| < c_{n+1}$.

and we know that the difference is less than $x^8/(8!)$ which allows us to control the error of the approximation.

The results of the computations using the various formulas are shown in the following table. The computations were aagain performed on a SUN SparcStation 10 in single precision Fortran.

x	$1-\cos$	$\sin^2/(1+\cos)$	Taylor
0.500000	0.122417	0.122417	0.122418
0.125000	$0.780231E-02$	$0.780233E-02$	$0.780233E-02$
$0.312500E-01$	$0.488222E-03$	$0.488241E-03$	$0.488242E-03$
$0.781250E-02$	$0.305176E-04$	$0.305174E-04$	$0.305174E-04$
$0.195312E-02$	$0.190735E-05$	$0.190735E-05$	$0.190735E-05$
$0.488281E-03$	$0.119209E-06$	$0.119209E-06$	$0.119209E-06$
$0.122070E-03$	0.	$0.745058E-08$	$0.745058E-08$
$0.305176E-04$	0.	$0.465661E-09$	$0.465661E-09$
$0.762939E-05$	0.	$0.291038E-10$	$0.291038E-10$
$0.190735E-05$	0.	$0.181899E-11$	$0.181899E-11$
$0.476837E-06$	0.	$0.113687E-12$	$0.113687E-12$
$0.119209E-06$	0.	$0.710543E-14$	$0.710543E-14$
$0.298023E-07$	0.	$0.444089E-15$	$0.444089E-15$

◇

Example 2.9 The roots of the quadratic equation $ax^2+bx+c=0$ are given by

$$x_\pm = \left(-b \pm \sqrt{b^2-4ac}\right)/(2a).$$

Let us consider the data $a = 5*10^{-4}, b = 100$, and $c = 5*10^{-3}$. The computed result (again using single precision Fortran on a SUN SparcStation 10) for the first root is

$$\hat{x}_+ = 0.$$

We know this result cannot be exact, since $x=0$ is a solution of the quadratic equation if and only if $c=0$.

The reason for this result is that $\sqrt{b^2-4ac} \approx b$ for the data given above. Thus we suffer from catastrophic cancellation.

A remedy is the following reformulation of the formula for x_+:

$$x_+ = \frac{-b \pm \sqrt{b^2-4ac}}{2a} = \frac{1}{2a}\frac{\left(-b+\sqrt{b^2-4ac}\right)\left(-b-\sqrt{b^2-4ac}\right)}{-b-\sqrt{b^2-4ac}} = \frac{1}{2a}\frac{4ac}{-b-\sqrt{b^2-4ac}}$$

Here the subtraction of two almost equal numbers is avoided and the computation using this formula gives $\hat{x}_+ = -0.5E-04$. ◇

Example 2.10 Consider the following computations (in Fortran on a SUN Sparcstation 10):

- Compute $r = p*p - 2*q*q$ with $p = 665857, q = 470832$. The results are:

using single precision, computed	r	$=$	0.
using double precision, computed	r	$=$	1.00000000000
true value	r	$=$	1.00000000000

- Compute $r = 9*p^4 - q^4 + 2*q^2$ with $p = 10864, q = 18817$. The results are:

using single precision, computed	r	$=$	708158976.000
using double precision, computed	r	$=$	2.00000000000
true value	r	$=$	1.00000000000

- Compute $r = p + q - p$ with $p = 10^{34}, q = -2$. The results are:

$$\begin{array}{lrcr} \text{using single precision, computed} & r & = & 0. \\ \text{using double precision, computed} & r & = & 0. \\ \text{true value} & r & = & -2. \end{array}$$

In all three computations catastrophic cancellation takes place. If double precision is used, then no, or fewer significant digits may be lost. However, usually catastrophic cancellation still takes place. The third computation is of particular interest. Sometimes people verify their computations by doing the computations first in single precision and then in double precision. If both results agree, then the result is assumed to be correct. The third computation shows that this is far from true! ◇

Example 2.11 We consider the computation of $e^{-5.5}$ using the series

$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!}.$$

The following computations are taken from [**?**, p.14]. Suppose that the floating point system is characterized by $\beta = 10$ and $t = 5$. The first few terms in the series for $e^{-5.5}$ are

$$\begin{array}{rcr} e^{-5.5} & = & 1.000 \\ & & -5.5000 \\ & & +15.125 \\ & & -27.730 \\ & & +38.129 \\ & & -41.942 \\ & & +38.446 \\ & & -30.208 \\ & & +20.768 \\ & & -12.692 \\ & & +6.9803 \\ & & -3.4902 \\ & & +1.5997 \\ & & \vdots \\ \hline & = & 0.0026363 \end{array}$$

The sum is terminated after 25 terms because subsequent terms no longer change it. The computed answer is 0.0026363. The exact answer is $e^{-5.5} \approx 0.00408677$. The reason for this poor result is that $\sum_{n=1}^{\infty} \frac{(-5.5)^n}{n!}$ is small (≈ 0.00408677) whereas $\sum_{n=1}^{\infty} \frac{(5.5)^n}{n!}$ is large. Notice the large terms at the beginning of the summation. These are much larger that the final sum.

A better way to compute $e^{-5.5}$ is

$$e^{-5.5} = \frac{1}{e^{5.5}} = \frac{1}{1.0000 + 5.5000 + 15.125 + \dots} = 0.0040865.$$

The computations are again done in five digit arithmetic. ◇

Example 2.12 In section 1.3 we have studied the solution of triangular systems. In particular, we have developed Algorithm 1.3 for the solution of upper triangular systems $Ux = b$ using back substitution. In exact arithmetic, Algorithm 1.3 determines whether a solution of $Ux = b$ exists and if one exists it returns a solution. What happens if we apply Algorithm 1.3 using floating point arithmetic?

For illustration we assume that the floating point system uses $\beta = 10$ and $t = 4$. The same effects occur if IEEE floating point arithmetic is used.

(i) Consider the system

$$\begin{pmatrix} 2 & 4 & 2 \\ 0 & 0 & 3 \\ 0 & 0 & 3 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 4 \\ 2 \\ 1 \end{pmatrix}.$$

It is easy to verify that this system does not have a solution.

The application of Algorithm 1.3, in ($\beta = 10$ and $t = 4$) floating point arithmetic yields

$$\bar{x}_3 = \mathsf{fl}(1/3) = 3.333 * 10^{-1}$$

and

$$0 * \bar{x}_2 = \mathsf{fl}(2 - \mathsf{fl}(3 * (3.333 * 10^{-1}))) = \mathsf{fl}(2 - 9.999 * 10^{-1}) = 1.$$

At this point Algorithm 1.3 would stop with the message that the system is not solvable.

(ii) Consider the system

$$\begin{pmatrix} 2 & 4 & 2 \\ 0 & 0 & 3 \\ 0 & 0 & 3 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 4 \\ 1 \\ 1 \end{pmatrix}.$$

This system has infinitely many solutions.

The application of Algorithm 1.3, in ($\beta = 10$ and $t = 4$) floating point arithmetic yields

$$\bar{x}_3 = \mathsf{fl}(1/3) = 3.333 * 10^{-1}$$

and

$$0 * \bar{x}_2 = 1 - \mathsf{fl}(3 * (3.333 * 10^{-1})) = 1 - 9.999 * 10^{-1} = 1.000 * 10^{-4}.$$

Again, Algorithm 1.3 would stop stop at this point with the message that the system is not solvable. Of course, this is not correct. ◇

Example 2.13 The derivative of a function $g : \mathbb{R} \to \mathbb{R}$ at x is defined by

$$g'(x) = \lim_{\delta \to 0} \frac{g(x+\delta) - g(x)}{\delta}.$$

A one-sided finite difference approximation of the derivative is given by

$$g'(x) \approx \frac{g(x+\delta) - g(x)}{\delta} \tag{2.17}$$

for a sufficiently small δ. Using the Taylor expansion

$$g(x+\delta) = g(x) + g'(x)\delta + \tfrac{1}{2} g''(x+\theta\delta)\delta^2$$

for some $\theta \in [0, 1]$, we express the approximation error as

$$g'(x) - \frac{g(x+\delta) - g(x)}{\delta} = -\tfrac{1}{2} g''(x+\theta\delta)\delta.$$

If we assume that $|g''(x+\theta\delta)| \le M$, for all $\theta \in [0,1]$, then

$$\left| g'(x) - \frac{g(x+\delta)-g(x)}{\delta} \right| \le \frac{M|\delta|}{2}. \tag{2.18}$$

We use (2.17) to approximate the derivative of $g(x) = \exp(x)$ at $x = 1$. The error $|\exp(1) - (\exp(1+\delta) - \exp(1))|/\delta$ for various $\delta > 0$ is shown in Figure 2.3. The computations were performed in MATLAB on a SUN Ultra 10 workstation using MATLAB Version 5.3.0.10183 (R11), Jan 21, 1999.

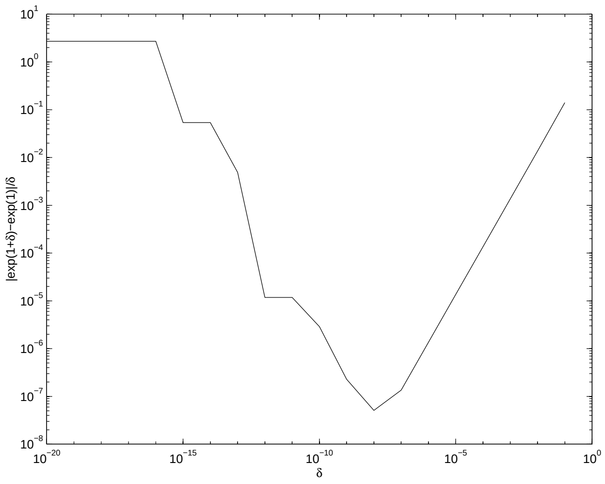

Figure 2.3: Finite Difference Error $|\exp(1) - (\exp(1+\delta) - \exp(1))|/\delta$.

Our error estimate (2.18) predicts that the finite difference error decreases linearly with the size of δ. This is not what we see in Figure 2.3. How can this discrepancy be explained? Our error estimate (2.18) assumes that the function values of g are available exactly. If we use floating point arithmetic this is not the case. Instead of the exact function g we only compute an approximation g_ϵ. In our case $g_\epsilon(1)$ is the floating point representation of $\exp(1)$ and $g_\epsilon(1+\delta)$ is the floating point representation of $\exp(1+\delta)$, i.e.,

$$g_\epsilon(1) = \mathsf{fl}(\exp(1)), \quad g_\epsilon(1+\delta) = \mathsf{fl}(\exp(\mathsf{fl}(1+\delta)))$$

Thus, the finite difference approximation of the derivative of g is not (2.17), but it is

$$g'(x) \approx \frac{g_\epsilon(x+\delta) - g_\epsilon(x)}{\delta}.$$

Suppose that

$$|g(x+\delta) - g_\epsilon(x+\delta)| \le \epsilon, \quad |g_\epsilon(x) - g(x)| \le \epsilon.$$

In our case

$$\begin{aligned} |\mathsf{fl}(\exp(\mathsf{fl}(1+\delta))) - \exp(1+\delta)| &\le \mathbf{u}|\exp(1+\delta)|, \\ |\mathsf{fl}(\exp(1)) - \exp(1)| &\le \mathbf{u}|\exp(1)|, \end{aligned}$$

i.e., $\epsilon \le \mathbf{u}|\exp(1+\delta)| \approx 10^{-15}$. From

$$\begin{aligned} & g'(x) - \frac{g_\epsilon(x+\delta) - g_\epsilon(x)}{\delta} \\ &= g'(x) - \frac{g(x+\delta)-g(x)}{\delta} + \frac{g(x+\delta) - g_\epsilon(x+\delta)}{\delta} + \frac{g_\epsilon(x) - g(x)}{\delta} \end{aligned}$$

and the estimate (2.18) we obtain

$$\begin{aligned}
&\left| g'(x) - \frac{g_\epsilon(x+\delta) - g_\epsilon(x)}{\delta} \right| \\
&\le \frac{M|\delta|}{2} + \frac{|g(x+\delta) - g_\epsilon(x+\delta)|}{\delta} + \frac{|g_\epsilon(x) - g(x)|}{\delta}, \\
&\le \frac{M|\delta|}{2} + \frac{2\epsilon}{|\delta|}. \qquad (2.19)
\end{aligned}$$

The first term, $M|\delta|/2$, of the error bound (2.19) results from the finite differences using exact function values and the second term $2\epsilon/|\delta|$ results from the inexact function values. The error bound as a function of δ is sketched in Figure 2.4. The $\delta > 0$ that minimizes the bound on the right hand side in (2.19) is given by

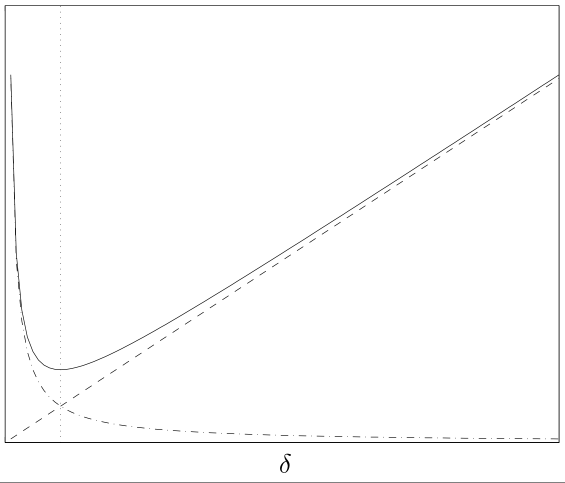

Figure 2.4: Finite Difference Error Bound.

$$\delta_* = (2/\sqrt{M})\,\sqrt{\epsilon}$$

and the corresponding error bound is

$$\frac{M|\delta_*|}{2} + \frac{2\epsilon}{|\delta_*|} = \left(\sqrt{M} + 4/\sqrt{M}\right)\,\sqrt{\epsilon}. \qquad (2.20)$$

Thus the quality of the finite difference derivative approximation is limited by the accuracy in function values. If, for example, the eerror in the function values is $\epsilon = 10^{-15}$, then (2.19), (2.20) indicate that we need to expect the error in the finite difference derivative approximation to be around 10^{-7} to 10^{-8}. To achieve this error in the finite difference derivative approximation, we have to choose the finite difference step size δ properly. If we choose δ much smaller than δ_*, then we will 'differentiate noise in the function evaluation' and $2\epsilon/|\delta|$ will dominate the error bound. If we choose δ much larger than δ_*, then the finite difference approximation of the exact function will not be good enough and $M|\delta|/2$ will dominate the error bound.

Note that the Figures 2.3 and 2.4 agree qualitatively. Figures 2.3 shows that the smallest error in the finite difference approximation is attained for δ around 10^{-7} to 10^{-8} and for those values of δ, the error in the finite difference approximation is around 10^{-7} to 10^{-8}, which agrees with (2.20). Also note that $\mathsf{fl}(1+\delta) = 1$ for $\delta < \mathbf{u}$. Thus for small δ,

$$\mathsf{fl}(\exp(\mathsf{fl}(1+\delta))) = \mathsf{fl}(\exp(1))$$

and the finite difference approximation is

$$\frac{\mathsf{fl}(\exp(\mathsf{fl}(1+\delta))) - \mathsf{fl}(\exp(1))}{\delta} = 0,$$

which explains why the error in Figures 2.3 is equal to exp(1) for small δ. $\diamond$