LECTURE 1: Number representation and errors

September 10, 2012

The representation of numbers in computers is usually not based on the decimal system, and therefore it is necessary to understand how to convert between different systems of representation. Computers also have a limited amount of bits for representing numbers, and for this reason numbers with arbitrarily large magnitude cannot be represented nor can floating-point numbers be represented with arbitrary precision. In this chapter, we define the key concepts of number representation in computers and discuss the main sources of errors in numerical computing.

0.1 Representation of numbers in different bases

0.1.1 Decimal system

We first review the representation of numbers in the familiar decimal system and then generalize the concept for systems with different base numbers.

Integer part

In the decimal system, the digits 0, 1, 2, 3, 4, 5, 6, 7, 8 and 9 are used for the representation of numbers. The individual digits in a number, such as 37294, are the coefficients of powers of 10:

$$\begin{aligned} 37294 &= 4 + 90 + 200 + 7000 + 30000 \\ &= 4 \times 10^0 + 9 \times 10^1 + 2 \times 10^2 + 7 \times 10^3 + 3 \times 10^4 \end{aligned}$$

In general, a string of digits represents a number according to the formula

$$a_n a_{n-1} \ldots a_1 a_0 = a_0 \times 10^0 + a_1 \times 10^1 + \ldots + a_{n-1} \times 10^{n-1} + a_n \times 10^n$$

Fractional part

A number between 0 and 1 is represented by a string of digits to the right of a decimal point. The individual digits represent the coefficients of negative powers of 10. In general, we have the formula

$$0.b_1 b_2 b_3 \ldots = b_1 \times 10^{-1} + b_2 \times 10^{-2} + b_3 \times 10^{-3} + \ldots$$

For example,

$$\begin{aligned} 0.7215 &= 7/10 + 2/100 + 1/1000 + 5/10000 \\ &= 7 \times 10^{-1} + 2 \times 10^{-2} + 1 \times 10^{-3} + 5 \times 10^{-4} \end{aligned}$$

General form of base-10 numbers

The representation of a real number in the decimal system is given by the following general form in which the integer part is the first summation and the decimal part is the latter summation.

$$(a_n a_{n-1} \ldots a_1 a_0 . b_1 b_2 b_3 \ldots)_{10} = \sum_{k=0}^{n} a_k 10^k + \sum_{k=1}^{\infty} b_k 10^{-k}$$

The notation $()_{10}$ is used for the decimal system to indicate that the number 10 is used as the base number.

0.1.2 Base β numbers

In computers, the number 10 is generally not used as the base for number representation. Instead, other systems, such as the *binary* (2 as the base), the *octal* (8 as the base), and the *hexadecimal* (16 as the base) systems are commonly used in computing. In the general form, the base is noted as β.

General form of base β numbers

The digits used in the general β representation are $0, 1, 2, \ldots, \beta - 1$. For example, in the octal representation of a number, the digits used are 0,1,2,3,4,5,6, and 7.

The general form of base β representation is given by

$$(a_n a_{n-1} \ldots a_1 a_0 . b_1 b_2 b_3 \ldots)_\beta = \sum_{k=0}^{n} a_k \beta^k + \sum_{k=1}^{\infty} b_k \beta^{-k}$$

The separator between the integer and fractional part is called the *radix point* since *decimal point* is reserved for base-10 numbers.

For example, in the octal representation, the individual digits before the radix point refer to increasing powers of 8:

$$\begin{aligned}(21467)_8 &= 7 + 6 \times 8 + 4 \times 8^2 + 1 \times 8^3 + 2 \times 8^4 \\ &= 7 + 8(6 + 8(4 + 8(1 + 8(2)))) \\ &= 9015\end{aligned}$$

To convert this number to the decimal system, we can build the nested form of the number by taking a common denominator and then replacing each of the numbers by its representation in the decimal system. We will soon discuss the conversion between different bases in more detail.

A number between 0 and 1, when expressed in the octal system, is given by the coefficients of negative powers of 8. For example,

$$\begin{aligned}(0.36207)_8 &= 3 \times 8^{-1} + 6 \times 8^{-2} + 2 \times 8^{-3} + 0 \times 8^{-4} + 7 \times 8^{-5} \\ &= 8^{-5}(7 + 8^2(2 + 8(6 + 8(3)))) \\ &= 15495/32768 = 0.42286987\ldots\end{aligned}$$

Conversion of integer part

We now consider a general procedure for converting from one base to another. We do this by considering the conversion of the integer and fractional parts separately.

Consider an integer N in the number system with base α:

$$N = (a_n a_{n-1} \ldots a_1 a_0)_\alpha = \sum_{k=0}^{n} a_k \alpha^k$$

In order to convert N to the number system with base β, we first write N in its nested form

$$N = a_0 + \alpha(a_1 + \alpha(a_2 + \ldots + \alpha(a_{n-1} + \alpha(a_n)) \ldots))$$

and then replace each of the numbers on the right by its representation in the β-arithmetic. The replacement requires a table showing how each of the numbers is represented in the

β-system. In addition, a base-β multiplication table may be required.

As an example, consider the conversion of the decimal number 3781 to binary form.

$$\begin{aligned}(3781)_{10} &= 1+10(8+10(7+10(3))) \\ &= (1)_2+(1010)_2((1000)_2+(1010)_2((111)_2+(1010)_2(11)_2)) \\ &= (111011000101)_2\end{aligned}$$

The following table can be used for the replacement of base-10 numbers by their binary counterparts:

DEC	BIN
0	0
1	1
2	10
3	11
4	100
5	101
6	110
7	111
8	1000
9	1001
10	1010

This calculations is easy for computers but quite tedious for humans. Therefore, another procedure can be used for hand calculations. Taking the conversion from the decimal to the binary system as an example, we continuously divide by 2, until 0 remains. Then the converted number is given by the remainders of the division in reversed order.

For example, here each number on the left hand side is obtained by dividing the previous number by 2 and taking the integer part, the quotient is marked on the right hand side. Thus the converted number is $(111011000101)_2$.

Quotients	Remainders
3781	
1890	1
945	0
472	1
236	0
118	0
59	0
29	1
14	1
7	0
3	1
1	1
0	1

Conversion of fractional part

The conversion of the fractional part can be done by the following direct, but somewhat useless approach:

$$\begin{aligned}(0.372)_{10} &= 3\times 10^{-1}+7\times 10^{-2}+2\times 10^{-3}\\ &= \frac{1}{(1010)_2}\left((011)_2+\frac{1}{(1010)_2}\left((111)_2+\frac{1}{(1010)_2}((010)_2)\right)\right)\end{aligned}$$

Since dividing binary arithmetic is not straightforward, we look for alternative ways of doing the conversion such that the arithmetic can be carried out in the decimal system.

Suppose that x is in the range $0<x<1$ and given in the β representation:

$$x=\sum_{k=1}^{\infty} c_k\beta^{-k}=(0.c_1c_2c_3\ldots)_\beta$$

Observe that

$$\beta x=(c_1.c_2c_3c_4\ldots)_\beta$$

because we only need to multiply by β to shift the radix point.

Thus the unknown digit c_1 can be described as the integer part of βx, denoted as $I(\beta x)$. Denote the fractional part of βx by $\mathcal{F}(\beta x)$. The process can be repeated until all the unknown digits c_k have been converted:

$$\begin{aligned} d_0 &= x & & \\ d_1 &= \mathcal{F}(\beta d_0) & c_1 &= I(\beta d_0)\\ d_2 &= \mathcal{F}(\beta d_1) & c_2 &= I(\beta d_1)\\ &\vdots & & \end{aligned}$$

For example, here we repeatedly multiply by 2 and remove the integer parts to get $(0.372)_{10} = (0.010111\ldots)_2$.

Products	Digits
0.372	
0.744	0
1.488	1
0.976	0
1.952	1
1.904	1
1.808	1
etc.	

0.2 Representation of numbers in computers

In this section, we discuss how numbers are represented in typical computers and how arithmetic between the numbers works. We begin by discussing integer numbers and then continue to floating-point numbers.

0.2.1 Integer numbers

A number in integer presentation is exact. Arithmetic between two integer numbers is also exact provided that (i) the answer is not outside the range of representable numbers, and that (ii) division is interpreted as producing an integer result (throwing any remainder away).

Unsigned integers are easy to represent. All digits are represented with bits 0 and 1, and thus the string of digits can be directly interpreted as a binary number. For example, with eight bits, we have

$$0011\,1001 = (1+8+16+32)_{10} = (57)_{10}$$

To include also negative numbers, we must assign a separate sign bit. The first bit of the string is the sign bit which is zero for positive numbers and one for negative numbers. The most often used method for obtaining the representation for negative numbers is a method called **two's complement**:

> In the binary representation of a positive number x, invert all the bits ($0 \leftrightarrow 1$), and add 1 to get the binary representation of $-x$.

For example, if we have eight bits for the representation (of which one is for the sign and seven for the digits), then

$$\begin{aligned} +2 &= 0000\,0010 \\ +1 &= 0000\,0001 \\ 0 &= 0000\,0000 \\ -1 &= 1111\,1111 \\ -2 &= 1111\,1110 \end{aligned}$$

The largest representable number in an n-bit system is $2^{n-1}-1$. For example, a 32-bit integer number can have values between

$$2^{31}-1 = 2147483647 = (01111111\ 11111111\ 11111111\ 11111111)_2 \quad \text{and}$$
$$-2^{31} = -2147483648 = (10000000\ 00000000\ 00000000\ 00000000)_2$$

In the minimum value, the sign bit has been interpreted as part of the number.

Most compilers do not give error messages of exceeding the range of integer numbers, except in some obvious situations. As an example, consider the following C code.

```
int main(void)
{
  short int si;
  int i, k, j1, j2, j3, j4;

  si = 1; i = 1;

  for(k=1; k<33; k++) {
    si *= 2;
    i *= 2;
    fprintf(stdout,"%3d%8d%12d\n",k,si,i);
 }

  j1 = -2147483647;
  fprintf(stdout,"\n%d\n",j1);
  j2 = -2147483648;
  fprintf(stdout,"\n%d\n",j2);
  j3 = 2147483647;
  fprintf(stdout,"\n%d\n",j3);
  j4 = 2147483648;
  fprintf(stdout,"\n%d\n",j4);
}
```

```
COMPILING:

warning: decimal constant is so large that it is unsigned
```

OUTPUT:

```
  1        2            2
  2        4            4
  3        8            8
  4       16           16
  5       32           32
  6       64           64
  7      128          128
  8      256          256
  9      512          512
 10     1024         1024
 11     2048         2048
 12     4096         4096
 13     8192         8192
 14    16384        16384
 15   -32768        32768
 16        0        65536
 17        0       131072
 18        0       262144
 19        0       524288
 20        0      1048576
 21        0      2097152
 22        0      4194304
 23        0      8388608
 24        0     16777216
 25        0     33554432
 26        0     67108864
 27        0    134217728
 28        0    268435456
 29        0    536870912
 30        0   1073741824
 31        0  -2147483648
 32        0            0

-2147483647

-2147483648

2147483647

-2147483648
```

0.2.2 Floating-point numbers

In a computer, there are no real numbers or rational numbers (in the sense of the mathematical definition), but all noninteger numbers are represented with finite precision.

For example, the numbers π or $1/3$ cannot be represented precisely (in any representation), and in some representations, the numbers 1.00000000 and 1.00000001 are equally large.

The floating-point representation in computers is based on an internal division of bits which are reserved for representing a given number x. The number is represented in three parts: a **sign** s that is either + or -, and an integer **exponent** c, and a positive **mantissa** M:

$$x = s \times B^{c-E} \times M$$

where B is the base of the representation (usually $B = 2$) and E is the bias of the exponent (a fixed integer constant for any given machine and representation which enables representing negative exponents without a separate sign bit).

In the decimal system, this corresponds to the following normalized floating-point form

$$x = \pm 0.d_1 d_2 d_3 \ldots \times 10^n = \pm r \times 10^n$$

where $d_1 \neq 0$ and n is an integer.

In most computers, floating-point numbers are represented in the following **standard IEEE floating-point** form

$$x = s \times 2^{c-E} \times (1.f)_2$$

The first bit s is the *sign bit* ($0 = +$ and $1 = -$). The next bits are used to represent the *exponent* c corresponding to 2^{c-E} where E is a constant which enables representing negative exponents without a separate sign bit. The last bits are reserved for the *mantissa* (also called the significand) which is given in the "1-plus" form $(1.f)_2$.

The IEEE standard defines *single-precision* (32 bits) and *double-precision* (64 bits) floating-point numbers. The available bits are allocated as shown in Fig. 1 (constant E is 127 for single precision and 1023 for double precision):

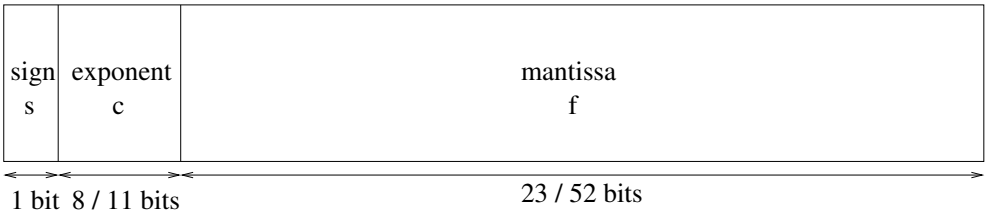

Figure 1: IEEE standard for floating-point numbers.

Machine numbers

A floating-point number system within a computer is always limited by the finite word-length of computers. This means that only a finite number of digits can be represented. As a consequence, numbers that are too large or too small cannot be represented. Moreover, most real numbers cannot be represented exactly in a computer. For example,

$$\frac{1}{10} = (0.1)_{10} = (0.0\,0011\,0011\,0011\,0011\,0011\ldots)_2$$

The effective number system for a computer is not a continuum but a discrete set of numbers called the **machine numbers**.

In a normalized representation (used in most computers), the bit patterns are as "left shifted" as possible, corresponding to the "1-plus" form of the mantissa (this form doesn't waste any bits but uses the correct exponent such that there are no leading zero bits in the mantissa). The machine numbers in the normalized representation are not uniformly spaced but unevenly distributed about zero.

For example, if we list all floating-point numbers which can be expressed in the following normalized form

$$x = \pm(0.1b_2b_3)_2 \times 2^{\pm k} \qquad (k, b_i \in [0,1])$$

where we have only two bits for the mantissa (b_2 and b_3) and one bit for the exponent (k), we get the set of numbers shown in Fig. 2. This shows the phenomenon known as the *hole at zero*. There is a relatively wide gap between the smallest positive number and zero, that is $(0.100)_2 \times 2^{-1} = 1/4$.

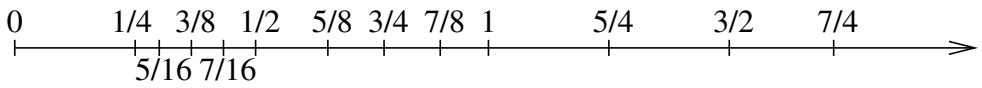

Figure 2: Representable numbers (machine numbers) in the example number system.

Machine epsilon

Arithmetic among numbers in floating-point representation is not exact even if the operands happen to be exactly represented. For example, two floating-point numbers are added by first right-shifting (dividing by two) the mantissa of the smaller one (in magnitude), simultaneously increasing the exponent, until the two operands have the same exponent. Low-order (least significant) bits of the smaller operand are lost by this shifting. If the two operands differ too much in magnitude, then the smaller operand is effectively replaced by zero. This leads us to the concept of machine accuracy.

Machine epsilon ε (or machine accuracy) is defined to be *the smallest number that can be added to 1.0 to give a number other than one*. Thus, the machine epsilon describes the accuracy of floating-point calculations. Note that this is not the same as the smallest floating-point number that is representable in a given computer (see below)!

When using single-precision numbers, there are 23 bits allocated for the mantissa. The machine epsilon is given by $\varepsilon = 2^{-23} \approx 1.19 \times 10^{-7}$, which means that numbers can be represented with approximately six accurate digits. In double precision, we have 52 bits allocated for the mantissa, and thus the machine epsilon is approximately $\varepsilon = 2^{-52} \approx 2.22 \times 10^{-16}$, giving the accuracy of about 15 digits.

To a great extent any arithmetic operation among floating-point numbers should be thought of as introducing an additional fractional error of at least ε. This type of error is called roundoff error (we return to this in more detail later in this chapter).

Smallest and largest numbers

The smallest and largest representable numbers are determined by the number of bits that are allocated for the exponent c. [Note: the machine epsilon, in contrast, depends on how many bits there are in the mantissa.]

The smallest representable number is given by

$$x_{min} = 2^{c_{min}}$$

and similarly, the largest number is

$$x_{max} = M_{max} \times 2^{c_{max}}$$

where c_{min} and c_{max} are the minimum and maximum values of the exponent, and M_{max} is the maximum value of the mantissa.

The value of c in the representation of single-precision floating-point numbers is restricted by the inequality

$$0 < c < (11\,111\,111)_2 = 255$$

The values 0 and 255 are reserved for special cases including ± 0 and $\pm\infty$. Hence, the actual exponent is restricted by

$$-126 < c - 127 < 127$$

Likewise, the mantissa is restricted by

$$1 \leq (1.f)_2 \leq (1.111\,111\,111\,111\,111\,111\,111\,11)_2 = 2 - 2^{-23}$$

The largest representable 32-bit number is therefore

$$(2 - 2^{-23})2^{127} \approx 2^{128} \approx 3.4 \times 10^{38}$$

The smallest number is

$$2^{-126} \approx 1.2 \times 10^{-38}$$

Zero, infinity and NaN

The IEEE standard defines some useful special characters.

The number **zero** is represented by all bits being zero. The sign bit, however, can obtain both values, and thus there are two different representations of zero: +0 (sign bit 0) and -0 (sign bit 1).

Infinity is defined by setting all the exponent bits to one and those in mantissa to zero. Depending on the sign bit, we have two different representation of infinity: $+\infty$ and $-\infty$. The following rules apply to arithmetic operations involving ∞:

$$\infty \pm x = \infty, \qquad x \times \infty = \infty, \qquad \frac{x}{\infty} = 0$$

In addition, the standard defines a constant called **NaN** (not a number) which is an error pattern rather than a number. It is obtained by setting all bits to ones.

C header file float.h

In C, the header file `float.h` contains constants which reflect the characteristics of that machine's floating-point arithmetic. Some of the most widely used values are (these are obtained from a Linux/Intel -machine):

```
Number of decimal digits for float (FLT_DIG)  = 6
Number of decimal digits for double (DBL_DIG) = 15

Precision for float (FLT_EPSILON)  = 1.19209e-07
Precision for double (DBL_EPSILON) = 2.22045e-16

Maximum float (FLT_MAX)  = 3.40282e+38
Maximum double (DBL_MAX) = 1.79769e+308

Minimum positive float (FLT_MIN)  = 1.17549e-38
Minimum positive double (DBL_MIN) = 2.22507e-308
```

Single or double precision?

Calculations with double-precision numbers are somewhat slower than when using single precision. The following loop ($N = 10^7$)

```
for(i=0; i<N; i++) {
  x += 0.1;
  y += sin(x);
  z += log(x)*cos(x);
}
```

took 5.70 seconds with single-precision numbers and 6.43 seconds with double-precision in a Linux/Intel machine (`y` and `z` were defined as `floats` or `doubles`). In this case, the difference is not large but may depend strongly on the machine in question.

In many cases, it is recommendable to use double-precision in order to avoid problems due to for example loss of significance or round-off errors.

0.3 Errors

Numerical calculations always involve approximations due to several reasons. These errors are not the result of poor thinking or carelessness (like programming errors) but they inevitably arise in all numerical calculations. We can divide the sources of errors roughly into four categories: model, method, initial values (data) and roundoff.

0.3.1 Modeling errors

When a practical problem is formulated into mathematical language, it is almost always necessary to make simplifications. Examples of modeling errors include leaving out less-influential factors (e.g., no air resistance in falling) or using a simplified description of a more complex system (e.g., classical description of a quantum-mechanical system).

Modeling errors are not discussed here in more detail but left as a subject of courses in the various application fields.

0.3.2 Methodological errors

The conversion of a mathematical problem into a numerical one is also a source of errors. Care should be taken to control these errors and to estimate their magnitude and thus the quality of the numerical solution. Note that by methodological errors we mean errors that would persist even if a hypothetical "perfect" computer had an infinitely accurate representation and no roundoff error. As a general rule, there is not much a programmer can do about the computer's roundoff error (see below for more details). Methodological errors, on the other hand, are entirely under the programmer's control. In fact, an incredible amount of work in the field of numerical analysis has been devoted to the fine minimization methodological errors!

An example of methodological errors is the **truncation error** (or chopping error) which is encountered when, for example, an interminating series is chopped:

$$x(t) = e^t = 1 + t + \frac{t^2}{2} + \ldots + \frac{t^n}{n!} + R_{n+1}(t)$$

Here the truncation error is: $-R_{n+1}(t)$. It follows that when n is sufficiently large, the excess term R_{n+1} is small and the chopping gives a good approximation of the exact result.

Another example of methodological errors is the **discretizing error** which results when a continuous quantity is replaced by a discrete approximation. For example, replacing the derivative by the difference quotient leads to a discretizing error:

$$x'(t) \approx y(t,h) = \frac{x(t+h) - x(t)}{h}$$

Or as another example, in numerical integration, the integrand is evaluated at a discrete set of points, rather than at "every" point.

0.3.3 Errors due to initial values

The initial values of a numerical computation can involve inaccurate values (e.g. measurements). When designing the algorithm, it is important to keep in mind that the initial errors must not accumulate during the calculation. There are also techniques for data filtering that are designed to decrease the effects of errors in the initial values.

0.3.4 Roundoff errors

Roundoff errors are the result of having a finite number of bits to represent floating-point numbers in computers. As already mentioned, arbitrarily large or small numbers cannot be represented and floating-point numbers cannot have arbitrary precision.

Consider a positive real number x in normalized floating-point form

$$x = (1.a_1a_2a_3\ldots a_{23}a_{24}a_{25}\ldots)_2 \times 2^m$$

The process of replacing x by its nearest machine number is called **correctly rounding**. The error involved is called the roundoff error. How large can it be?

One nearby machine number can be obtained by rounding down or by dropping the excess bits $a_{24}a_{25}\ldots$ (if only 23 bits have been allocated to the mantissa). This machine number is

$$x_- = (1.a_1a_2a_3\ldots a_{23})_2 \times 2^m$$

Another nearby machine number is found by rounding up. It is found by adding one unit to a_{23} in the expression for x_-. Thus,

$$x_+ = [(1.a_1a_2a_3\ldots a_{23})_2 + 2^{-23}] \times 2^m$$

The closer of these machine numbers is chosen for x.

Since the unit roundoff error (machine epsilon) for a 32-bit floating-point representation is $\varepsilon = 2^{-23}$, we notice that in the case of rounding to the nearest, the relative error is no greater than $\varepsilon/2$. This is illustrated in Fig. 3. [Remember that the machine epsilon is defined to be the smallest number such that when added to 1, the result is not equal to 1.]

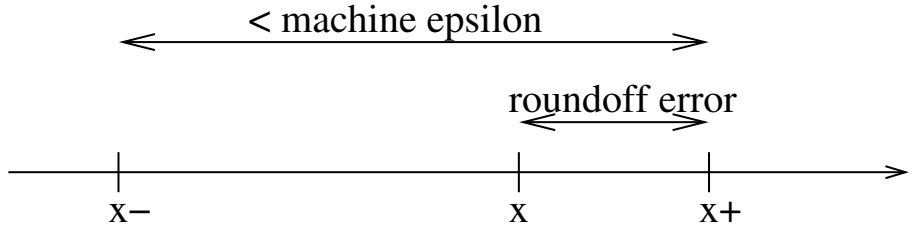

Figure 3: Rounding to the nearest: the number x is represented by the machine number that lies closest to the true value of x. The roundoff error involved in the operation is no greater than $\varepsilon/2$.

0.3.5 Elementary arithmetic operations

We continue to examining errors that are produced in basic arithmetic operations. Round-off errors accumulate with increasing amounts of calculation. If you perform N arithmetic operations to obtain a given value, the total roundoff error might be of the order $\sqrt{N}\varepsilon$, if you are lucky! [The square root comes from a random walk, assuming that the errors come in randomly up or down.]

However, this estimate can be badly off the mark for two reasons:
(i) It frequently happens that for some reason (because of the regularities of your calculation or the peculiarities of your computer) the roundoff errors accumulate preferentially in one direction. In this case, the total error will be of order $N\varepsilon$.
(ii) Some especially unfavorable operations can increase the roundoff error substantially in a single arithmetic operation. A good example is the subtraction of two almost equal numbers, which results in the loss of significance (see the discussion later in this chapter).

To discuss these issues in more detail, we first introduce the notation **fl**(x) to denote the *floating-point machine number* that corresponds to the real number x.

For example, in a hypothetical five-decimal machine,

$$\mathrm{fl}(0.3721871422 \times 10^4) = 0.37219 \times 10^4$$

If x is any real number within the range of the computer, then the error involved in the operation is bound by $\varepsilon/2$ (assuming correct rounding)

$$\frac{|x - \mathrm{fl}(x)|}{|x|} \leq \frac{1}{2}\varepsilon$$

This can be written as

$$\mathrm{fl}(x) = x(1+\delta) \qquad (|\delta| \leq \frac{1}{2}\varepsilon)$$

Remember that the unit round-off error or the machine epsilon is the smallest positive machine number ε such that

$$\mathrm{fl}(1+\varepsilon) > 1$$

Let the symbol $\odot$ denote any one of the operations, +, - , $\times$, or $\div$. Suppose that whenever two *machine numbers* x and y are combined arithmetically, the computer will produce $\mathrm{fl}(x \odot y)$ instead of $x \odot y$. Under this assumption, the previous analysis gives

$$\mathrm{fl}(x \odot y) = (x \odot y)(1+\delta) \qquad (|\delta| \leq \frac{1}{2}\varepsilon)$$

It is of course possible that $x \odot y$ overflows or underflows, but except for this case, the above assumption is realistic for most computers.

However, in real calculations, the initial values are given as *real numbers* and not as machine numbers. For example, the relative error for addition can be estimated by

$$\begin{aligned} z &= \mathrm{fl}[\mathrm{fl}(x) + \mathrm{fl}(y)] \\ &= [x(1+\delta_1) + y(1+\delta_2)](1+\delta_3) \\ &\approx (x+y) + x(\delta_1+\delta_3) + y(\delta_2+\delta_3) \end{aligned}$$

The relative round-off error is

$$\left|\frac{(x+y)-z}{(x+y)}\right| = \left|\frac{x(\delta_1+\delta_3)+y(\delta_2+\delta_3)}{(x+y)}\right|$$
$$= \left|\delta_3 + \frac{x\delta_1+y\delta_2}{(x+y)}\right|$$

We notice that this *cannot be bounded*, because the second term has a denominator that can be zero or close to zero. Problems due to these type of situations are discussed next.

0.4 Loss of significance

In this section, we discuss how the problem of loss of significance arises and how it can be avoided by various techniques, such as the use of rationalization, Taylor series, trigonometric identities, and so on.

0.4.1 Significant digits

We begin by considering the concept of significant digits in a number. Consider a real number x expressed in normalized scientific form. For example,

$$x = 0.3721498 \times 10^{-5}$$

The digits of x do not have the same significance since they represent different powers of 10. Thus, 3 is the *most significant digit* and 8 is the *least significant digit*.

A *mathematically exact* quantity, such as π, can be expressed with as many significant digits as we like. A *measured* quantity, however, always involves an error whose magnitude depends on the measuring device. Likewise, all *numerically computed* quantities are not exact, but can only be expressed with a certain precision depending on the machine.

It is important to understand that the precision of a computed quantity is determined by the *least precise* value that was used in the computation.

For example, suppose we have a measured quantity $s = 0.736$. It is a scientific convention that the least significant digit given in a measured quantity should be in error by at most five units. The following computation gives

$$y = s \times \sqrt{(2)} \approx 0.1040861182 \times 10^1$$

but the result should be reported as 0.104×10^1.

Remember that the output of a computer program often produces numbers with a long list of digits. It is perfectly acceptable (and even recommended) to include these in the raw output, but after analysis, the final results should always be given with appropriate precision!

0.4.2 Loss of significance

In some cases, the relative error involved in arithmetic calculations can grow significantly large. This often involves subtracting two almost equal numbers.

For example, consider the case $y = x - \sin(x)$. Let us have a computing system which works with ten decimal digits. Then

$$\begin{aligned} x &= 0.66666\,66667 \times 10^{-1} \\ \sin x &= 0.66617\,29492 \times 10^{-1} \\ x - \sin x &= 0.00049\,37175 \times 10^{-1} \\ &= 0.49371\,75000 \times 10^{-4} \end{aligned}$$

Thus the number of significant digits was reduced by three! Three **spurious zeros** were added by the computer to the last three decimal places, but these are not significant digits. The correct value is $0.49371\,74327 \times 10^{-4}$.

The simplest solution to this problem is to use floating-point numbers with higher precision, but this only helps up to a certain point. A better approach is to anticipate that a problematic situation may arise and change the algorithm in such a way that the problem is avoided.

0.4.3 Loss of precision theorem

Exactly how many significant binary digits are lost in the subtraction $x - y$ when x is close to y?

Let x and y be normalized floating-point numbers with $x > y > 0$.
If $2^{-p} \leq 1 - y/x \leq 2^{-q}$ for some positive integers p and q, then at most p and at least q significant binary digits are lost in the subtraction $x - y$.

Example.
How many significant bits are lost in the subtraction $x - y = 37.593621 - 37.584216$?

We have

$$1 - \frac{y}{x} = 0.0002501754$$

This lies between $2^{-12} = 0.000244$ and $2^{-11} = 0.000488$. Hence, at least 11 but at most 12 bits are lost.

0.4.4 Avoiding loss of significance

i. Rationalizing

Consider the function

$$f(x) = \sqrt{(}x^2+1)-1$$

We see that near zero, there is a potential loss of significance.

However, the function can be rewritten in the form

$$f(x) = (\sqrt{(}x^2+1)-1)\left(\frac{\sqrt{(}x^2+1)+1}{\sqrt{(}x^2+1)+1}\right)$$
$$= \frac{x^2}{\sqrt{(}x^2+1)+1}$$

This allows terms to be canceled out and therefore removes the problematic subtraction.

ii. Using series expansion

Consider the function

$$f(x) = x - \sin x$$

whose values are required near $x = 0$. We can avoid the loss of significance by using the Taylor series for $\sin x$

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} - \dots$$

For x near zero, the series converges quite rapidly.

We can now rewrite the function f as

$$f(x) = x - \left(x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} - \dots\right) = \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} - \dots$$

This is a very effective form for calculating f for small x.

iii. Using trigonometric identities

As a simple example, consider the function

$$f(x) = \cos^2(x) - \sin^2(x)$$

There will be loss of significance at $x = \pi/4$.

The problem can be solved by the simple substitution

$$\cos^2(x) - \sin^2(x) = cos(2x)$$

0.4.5 Range reduction

Another cause for loss of significant digits is the use of various library functions with very large arguments.

For example, consider the sine function whose basic property is its periodicity

$$\sin(x) = \sin(x + 2\pi n)$$

for all real values of x and all integer values of n.

The periodicity is used in the the computer evaluation of $\sin x$: One only needs to know the values of $\sin x$ in some fixed interval of length 2π in order to compute $\sin x$ for arbitrary x. This is called **range reduction**.

This procedure leads to an unavoidable loss of precision if the original argument x is very large. For example, if we want to evaluate $\sin(12532.14)$, we first subtract 3988π from the original argument and calculate the value of $\sin(3.47)$. Here the argument has only three significant figures due to the subtraction! Thus our computed value of $\sin(12532.14)$ can only be given with three significant digits!

The only way to improve the situation is to use **double- or extended-precision** programming. This is recommended for variables which are used as arguments of library functions such as the sine function.