Fundamentals of Simulation Methods

Exercise Sheet 1 — Issues of floating point arithmetic

Daniel Rosenblüh, Janosh Riebesell · Lecturer: Volker Springel · October 23rd, 2015

1 Machine epsilon

Write a computer program in C, C++, or Python that experimentally determines the machine epsilon ϵ_m, i.e. the smallest number ϵ_m such that $1 + \epsilon_m$ still evaluates to something different from 1, for the following data types:
(a) `float`,
(b) `double`,
(c) `long double`.

> **Note:** The significance of ϵ_m lies in its ability to establish an upper bound on the relative error due to rounding in floating point arithmetic, where rounding denotes the process of finding the closest match to any real number $r \in \mathbb{R}$ within a floating point system.[1]

The simplest way to find the value of ϵ_m in C and C++ for different data types is to consult the macro constants `FLT_EPSILON`, `DBL_EPSILON`, and `LDBL_EPSILON` defined by the `float.h` header. They can be printed to the console via

```
printf("according to <float.h> header:\n\tfloat epsilon = \t%e\n\tdouble epsilon =
\t%e\n\tlong double epsilon = \t%Le\n", FLT_EPSILON, DBL_EPSILON, LDBL_EPSILON);
```

By way of calculating these values ourselves, we can use the simple algorithm

```
float f_epsilon = 1;
while (1 + f_epsilon/2 != 1) {
    f_epsilon /= 2;
}
```

and equivalently for `double` and `long double`. This approximate solution yields

$$\epsilon_{m,f} = 1.192\,093 \times 10^{-7}, \qquad \epsilon_{m,d} = 2.220\,446 \times 10^{-16}, \qquad \epsilon_{m,ld} = 1.084\,202 \times 10^{-19}. \tag{1}$$

which coincides to given precision with the corresponding `float.h` macro constants.

2 Pitfalls of floating point arithmetic

Consider the following numbers:

```
double a = 1.0e17;
double b = -1.0e17;
double c = 1.0;
double x = (a + b) + c;
double y = a + (b + c);
```

Calculate the results for `x` and `y`. Which one is correct, if any? Explain, why the law of associativity is broken here.

[1] Notes are written out of personal interest and not relevant for the solution of the exercise.

`x` will give the correct answer of 1.0. When evaluating `y`, a `double` does not have sufficient accuracy to correctly perform the sum of `b` and `c` since

$$1 = |c| < \epsilon_{m,d} \cdot |b| \approx 2.22 \times 10^{-16} \times 10^{17} \approx 22. \tag{2}$$

The value of c will be lost to machine precision and y evaluates to 0.0.

3 Pitfalls of floating point representation

Consider the following C/C++ code:

```
float   x = 0.01;
double  y = x;
double  z = 0.01;
int i=x*10000;
int j=y*10000;
int k=z*10000;
printf("%d  %d  %d\n", i, j, k);
```

which prints out three integer numbers.

(a) Explain why the numbers are not all equal.

(b) Determine the rational number n/m, where n and m are natural numbers, that is represented by the single-precision IEEE-754 floating point variable `x` in the above example. Note: This number is not 1/100.

(a) The above code snippet prints `100 99 100`. The reason is that while the fraction 1/100 has a simple enough representation as a decimal number, it cannot be represented exactly in binary format.

$$1/100 = (0.01)_{10} = \left(0.0000001010001111010111\overline{}\right)_2 \tag{3}$$

Writing

```
#include <stdio.h>
int main() {
float x = 0.01;
printf("%.29f", x);}
```

reveals that the stored value of `x` is 0.009 999 999 776 482 582 092 285 156 25. This value is then assigned to `y` without further loss of precision since `double` has a higher precision than `float`. This same higher precision allows `y` to carry along the rounding error it received from `x` through the multiplication by 10 000, whereas the `float x` incurs another rounding error compensating the first. Multiplied by 10 000, `x` and `y` become 100.0 and 99.999 997 764..., respectively. This result is then cast to type `int`, where digits after the floating point are simply truncated. This explains the values of 100 and 99 for `i` and `j`.

When assigning 0.01 to the `double z`, we produce less of a rounding error right from the start. However, `z` still has finite precision and so is similarly unable to store 0.01 exactly. Its internal value is 0.010 000 000 000 000 000 208 166.... The error now overestimates so when casting `z` to type `int`, we simply get 100 again.

(b) By writing `x` as $\frac{x}{1}$, expanding the fraction by 10^{29} and reducing as far as possible, we get the rational representation of `x`:

$$\texttt{x} = \frac{5368709}{536870912}. \tag{4}$$

4 Packing of numbers

Estimate how many numbers there are in the interval between 1.0 and 2.0, and in between the interval of 511.0 to 512.0, for IEEE-754 numbers with
(a) single precision.
(b) double precision.

The spacing between numbers in floating-point format depends on their implementation.

(a) IEEE-754 numbers with single precision are 32 bit in size, of which 1 bit, the first, is the sign bit, followed by 8 exponent bits, and another 23 fraction bits to encode the actual numbers after the floating point. (The IEEE standard prescribes a so-called *normalized significand*, which requires the number in front of the point to be 1 with the exponent adjusted accordingly.) Thus, with an exponent of zero, we can (including the sign bit) represent 2^{24} different numbers, exactly half of which, the positive ones, lie in the interval $[1, 2)$. (That makes for a spacing of 2^{-23} in the interval $[1, 2)$ for IEEE-754 single-precision numbers.)

    To represent numbers in the interval $[511, 512)$, the exponent bits encode the decimal 8. Therefore, number spacing is scaled by a factor of 2^8 and we estimate only $2^{23}/2^8 = 2^{15}$ numbers within $[511, 512)$.

(b) IEEE-754 numbers with double precision are 64 bit in size, of which again the first is the sign bit, followed by 11 exponent bits, and 52 bits for the mantissa.

    Thus, there are 2^{52} numbers in the interval $[1, 2)$ and 2^{44} in $[511, 512)$.

5 Summing a long list of numbers

On the lecture's moodle-site, you'll find a binary file `numbers.dat` (8 MB). This contains first a 32 bit integer number that gives the number of double-precision values stored in the file (10^6 in the provided example), followed by the numbers themselves.
Write a read-statement for these numbers, and then try to sum them up, using different approaches.

(a) First, sum the numbers with a simple loop, sequentially from beginning to end. Write down the result.
(b) Next, sum them from end to beginning, reversing the initial direction of the loop. Write down the result.
(c) Sort the numbers by their magnitude, and sum them from small to large. What do you get now?
(d) Repeat the last experiment by using a summation variable of type `long double`. Do you think the obtained result is trustworthy and correct? (Optional: Try to get real 'quad-double' precision to work where the machine epsilon ϵ_m is of the order of 10^{-34} or so. Check the manual of your compiler what `long double` actually stands for and how you can switch on real 128 bit floating point precision emulated in software.)
(e) Write a program that exactly sums the list of numbers using the GMP library (GNU big number library) that allows arbitrary precision floating point accuracy. What do you get?

(a) A sequential (forward) sum of all values stored in `numbers.dat`, except the first one, yields $\Sigma_{\text{for}} = -6\,516\,239\,353\,685\,426$.
(b) Summing backwards, on the other hand, gives $\Sigma_{\text{back}} = 96\,953\,142\,435\,211\,540\,151\,926\,784$.
(c) Sorting the numbers in `numbers.dat` and summing again, we get $\Sigma_{\text{sort}} = 0$.
(d) Using `long double` precision to do the sorted sum, we still get $\Sigma_{\text{long}} = 0$.
(e) Finally, bringing out the heavy machinery and performing the sorted sum with 512 bit-precision, we obtain the confidence inspiring result $\Sigma_{\text{gmp}} = 42$.