Why and how to use arbitrary precision

Kaveh R. Ghazi Vincent Lefèvre Philippe Théveny Paul Zimmermann

Most nowadays floating-point computations are done in double precision, i.e., with a significand (or mantissa, see the "Glossary" sidebar) of 53 bits. However, some applications require more precision: double-extended (64 bits or more), quadruple precision (113 bits) or even more. In an article published in *The Astronomical Journal* in 2001, Toshio Fukushima says: "*In the days of powerful computers, the errors of numerical integration are the main limitation in the research of complex dynamical systems, such as the long-term stability of our solar system and of some exoplanets [...]*" and gives an example where using double precision leads to an accumulated round-off error of more than 1 radian for the solar system! Another example where arbitrary precision is useful is static analysis of floating-point programs running in electronic control units of aircrafts or in nuclear reactors.

Assume we want to determine 10 decimal digits of the constant $173746a + 94228b - 78487c$, where $a = \sin(10^{22})$, $b = \log(17.1)$, and $c = \exp(0.42)$. We will consider this as our running example throughout the paper. In this simple example, there are no input errors, since all values are known exactly, i.e., with infinite precision.

Our first program — in the C language — is:

```
#include <stdio.h>
#include <math.h>

int main (void)
{
  double a = sin (1e22), b = log (17.1);
  double c = exp (0.42);
  double d = 173746*a + 94228*b - 78487*c;
  printf ("d = %.16e\n", d);
  return 0;
}
```

and we get (all experiments in this paper are done with GCC 4.3.2 running on a 64-bit Core 2 under Fedora 10, with GNU LIBC 2.9):

```
d = 2.9103830456733704e-11
```

This value is completely wrong, since the expected result is $-1.341818958 \cdot 10^{-12}$. We can change `double` into `long double` in the above program, to use double-extended precision (64-bit significand) on this platform[1] — and change `sin(1e22)` to `sinl(1e22L)`, `log` to `logl`, `exp` to `expl`, `%.16e` into `%.16Le`; we then get:

```
d = -1.3145040611561853e-12
```

This new value is "almost as wrong" as the first one. Clearly the working precision is not enough.

1 What can go wrong

Several things can go wrong in our running example. First constants such as `1e22`, `17.1` or `0.42` might not be exactly representable in binary. This problem should not occur for the constant `1e22`, which is exactly representable in double precision, assuming the compiler transforms it into the correct binary constant, as required by IEEE 754 (see the IEEE 754 sidebar). However `17.1` cannot be represented exactly in binary, the closest double-precision value is $2406611050876109 \cdot 2^{-47}$, which differs by about $1.4 \cdot 10^{-15}$. The same problem happens with `0.42`.

Secondly for a mathematical function, say sin, and a floating-point input, say $x = 10^{22}$, the value

[1] On ARM computers, `long double` is double precision only; on Power PC, it corresponds to double-double arithmetic (see the "Glossary" sidebar), and under Solaris, to quadruple precision.

$\sin x$ is usually not exactly representable as a double-precision number. The best we can do is to round $\sin x$ to the nearest double-precision number, say y. In our case we have $y = -7675942858912663 \cdot 2^{-53}$, and the error $y - \sin x$ is about $6.8 \cdot 10^{-18}$.

Thirdly, IEEE 754 requires neither *correct rounding* (see the IEEE 754 sidebar) of the mathematical functions like $\sin, \log, \exp$, nor even some given accuracy, and results are completely platform-dependent [3]. However, while the 1985 version did not say anything about these mathematical functions, correct rounding became recommended in the 2008 revision. Thus the computed values for the variables `a`, `b`, `c` might differ from several ulps (units in last place, see the "Glossary" sidebar) from the correctly-rounded result. On this particular platform, whether optimizations are enabled or not — see Section 3 — all three functions are correctly rounded for the corresponding binary arguments x, which are themselves rounded with respect to the decimal inputs.

Finally a cancellation happens in the sum $173746a + 94228b - 78487c$. Assuming it is computed from left to right, the sum $173746a + 94228b$ is rounded to $x = 1026103735669971 \cdot 2^{-33} \approx 119454.19661583972629$, while $78487c$ is rounded to $y = 4104414942679883 \cdot 2^{-35} \approx 119454.19661583969719$. By Sterbenz's theorem (see the "Glossary" sidebar), there are no round-off errors when computing $x - y$; however the accuracy of the final result is clearly bounded by the round-off error made when computing x and y, i.e., $2^{-36} \approx 1.5 \cdot 10^{-11}$. Since the exact result is of the same order of magnitude, this explains why our final result d is completely wrong.

2 The GNU MPFR library

By *arbitrary precision*, we mean the ability for the user to choose the precision of each calculation. (One also says *multiple precision*, since this means that large significands (see the "Glossary" sidebar) are split over several machine words; modern computers can store at most 64 bits in one word, i.e., about 20 digits.) Several programs or libraries enable one to perform computations in arbitrary precision, in particular most computer algebra systems, like Mathematica, Maple, or Sage. We focus here on GNU MPFR, which is a C library dedicated to floating-point computations in arbitrary precision (for other languages than C, see the "Other languages" sidebar). What makes MPFR different is that it guarantees correct rounding (see the IEEE 754 sidebar). With MPFR, our running example becomes:

```
#include <stdio.h>
#include <stdlib.h>
#include "mpfr.h"

int main (int argc, char *argv[])
{
  mp_prec_t p = atoi (argv[1]);
  mpfr_t a, b, c, d;
  mpfr_inits2 (p, a, b, c, d, (mpfr_ptr) 0);
  mpfr_set_str (a, "1e22", 10, GMP_RNDN);
  mpfr_sin (a, a, GMP_RNDN);
  mpfr_mul_ui (a, a, 173746, GMP_RNDN);
  mpfr_set_str (b, "17.1", 10, GMP_RNDN);
  mpfr_log (b, b, GMP_RNDN);
  mpfr_mul_ui (b, b, 94228, GMP_RNDN);
  mpfr_set_str (c, "0.42", 10, GMP_RNDN);
  mpfr_exp (c, c, GMP_RNDN);
  mpfr_mul_si (c, c, -78487, GMP_RNDN);
  mpfr_add (d, a, b, GMP_RNDN);
  mpfr_add (d, d, c, GMP_RNDN);
  mpfr_printf ("d = %1.16Re\n", d);
  mpfr_clears (a, b, c, d, NULL);
  return 0;
}
```

This program takes as input the working precision p. With $p = 53$, we get:

```
d = 2.9103830456733704e-11
```

Note that this is exactly the result we got with double precision. With $p = 64$, we get:

```
d = -1.3145040611561853e-12
```

which matches the result we got with double-extended precision. With $p = 113$, which corresponds to IEEE-754 quadruple precision, we get here:

```
d = -1.3418189578296195e-12
```

which matches exactly the expected result.

3 Constant folding

In a given program, when an expression is a constant like $3 + (17 \times 42)$, it might be replaced at compile-time by its computed value. The same holds for floating-point values, with an additional difficulty: the compiler should be able to determine the rounding mode to use. This replacement done by the compiler is known as *constant folding* [4]. With correctly-rounded constant folding, the generated constant depends only on the format of the floating-point type on the target platform, and no more on the processor, system, and mathematical library used by the building platform. This provides both *correctness* (the generated constant is the correct one with respect to the precision and rounding mode) and *reproducibility* (platform-dependent issues are eliminated). As of version 4.3, GCC uses MPFR to perform constant folding of intrinsic (or builtin) mathematical functions such as $\sin, \cos, \log, \mathrm{sqrt}$. Consider for example the following program:

```
#include <stdio.h>
#include <math.h>

int main (void)
{
  double x = 2.522464e-1;
  printf ("sin(x) = %.16e\n", sin (x));
  return 0;
}
```

With GCC 4.3.2, if we compile this program without optimizing (i.e., using `-O0`), we get as result `2.4957989804940914e-01`. With optimization (i.e., using `-O1`), we get `2.4957989804940911e-01`. Why this discrepancy? With `-O0`, the expression `sin(x)` is evaluated by the mathematical library (here the GNU C library, also called GNU libc or glibc). With `-O1`, GCC recognizes the expression `sin(x)` is a constant, with rounding mode is to nearest, calls MPFR to evaluate it, and directly replaces `sin(x)` with its correctly rounded value.[2] The correct value is the one obtained with `-O1`. Note however that if the GNU C library does not round correctly on that example, most values are correctly rounded by the GNU C library (on computers without extended precision), as recommended by IEEE 754-2008. In the future, we can hope correct rounding for every input and every function.

Note: on x86 processors, the GNU C library uses the `fsin` implementation from the x87 co-processor, which for $x = 0.2522464$ returns the correctly rounded result. However this is just by chance, since among the 10^7 double-precision numbers including 0.25 and above, `fsin` gives an incorrect rounding for 2452 of them.

4 Conclusion

We have seen in this paper that using double precision variables with a significand of 53 bits can lead to much less than 53 bits of accuracy in the final results. Among the possible reasons for this loss of accuracy are roundoff errors, numerical cancellations, errors in binary-decimal conversions, bad numerical quality of mathematical functions, ... We have seen that using arbitrary precision, for example with the GNU MPFR library, helps to increase the final accuracy. More importantly, the correct rounding of mathematical functions in MPFR helps to increase the reproducibility of floating-point computations among different processors, with different compilers and/or operating systems, as demonstrated by the example of constant folding within GCC.

References

[1] Fousse, L., Hanrot, G., Lefèvre, V., Pélissier, P., and Zimmermann, P. MPFR: A multiple-precision binary floating-point library with correct rounding. *ACM Trans. Math. Softw. 33*, 2 (2007), article 13.

[2] When compiling with `-O1`, we can even omit linking with the mathematical library, i.e., `gcc -O1 test.c`, which proves that the mathematical library is not used at all. On the contrary, `gcc -O0 test.c` yields a compiler error, and `gcc -O0 test.c -lm` works, showing that the mathematical library is needed here. To disable constant folding and other optimizations on intrinsic builtin functions one can use `gcc -fno-builtin`, or more specifically `gcc -fno-builtin-sin` to target the sin function by itself.

[2] IEEE standard for floating-point arithmetic. ANSI-IEEE standard 754-2008, 2008. Revision of ANSI-IEEE Standard 754-1985, approved June 12, 2008: IEEE Standards Board.

[3] Lefèvre, V. Test of mathematical functions of the standard C library. http://www.vinc17.org/research/testlibm/.

[4] Wikipedia. Constant folding. http://en.wikipedia.org/wiki/Constant_folding.

The IEEE 754 standard (sidebar)

IEEE 754 is a widely used standard for floating-point representations and operations (your computer uses it every day without you being aware of it). It is very important because it defines floating-point *formats* — enabling two computers to exchange floating-point values without any loss of accuracy — and it requires *correct rounding* for arithmetic operations, which guarantees that the same program will yield identical results on two different computers[3]. IEEE 754 was first approved in 1985, and was revised in 2008 [2]. We describe here this revision, denoted as IEEE 754-2008. IEEE 754-2008 defines both *basic formats* — for computations — and *interchange formats* — to exchange data between different implementations. There are five basic formats: `binary32`, `binary64`, `binary128`, `decimal64` and `decimal128`. The `binary32` and `binary64` yield single and double (binary) precision respectively, and usually correspond to the `float` and `double` data-types in the ISO C language. The decimal formats are new to IEEE 754-2008; some preliminary support is available in GCC. For example `decimal64` is denoted by `_Decimal64` in GCC, in conformance with the current draft on decimal floating-point arithmetic in C, TR 24732[4]. Our running example becomes then:

```
#include <stdio.h>
#include <math.h>

int main (void)
{
  _Decimal64 a = sin (1e22);
  _Decimal64 b = log (17.1);
  _Decimal64 c = exp (0.42);
  _Decimal64 d = 173746*a+94228*b-78487*c;
  printf ("d = %.16e\n", (double) d);
  return 0;
}
```

and we get:

```
d = 0.0000000000000000e+00
```

(Note that we had to convert the final result d to `double` since the GNU C library does not yet support printing of decimal formats.)

IEEE 754 requires *correct rounding* for the four basic arithmetic operations ($+, -, \times, \div$), the square root, and the radix conversions (for example when reading a decimal string into a binary format, or when printing a binary format into a decimal string). This means that the computed result should be as if computed in infinite precision, and then rounded with respect to the current *rounding mode*. IEEE 754-2008 specifies five rounding modes (or *attributes*): `roundTowardPositive`, `roundTowardNegative`, `roundTowardZero`, `roundTiesToEven`, and `roundTiesToAway`.

Double-extended precision and Linux. The traditional floating-point unit of the 32-bit x86 processors can be configured to round the results either in double precision (53-bit significand) or in extended precision (64-bit significand). Most operating systems, such as FreeBSD, NetBSD and Microsoft Windows, chose to configure the processor so that, by default, it rounds in double precision. On the other hand, under Linux, the rounding is done in extended precision. This is a bad choice for the reasons detailed in `http://www.vinc17.org/research/extended.en.html`.

[3] Under some conditions that we omit here.

[4] `http://www.open-std.org/jtc1/sc22/wg14/www/projects`

Glossary (sidebar)

Radix, significand, and exponent. If x is a floating-point number of precision p in radix β, it can be written $x = \pm 0.d_1 d_2 \dots d_p \cdot \beta^e$, where $s = \pm 1$ is the *sign* of x, $m = 0.d_1 d_2 \dots d_p$ is the *significand* of x, and e is the exponent of x. Note that this representation is not unique, unless we force d_1 to be non-zero. Note also that different conventions are possible for the significand, which lead to different values for the exponent. For example, IEEE 754-2008 uses $m = d_1.d_2 \dots d_p$, which gives an exponent smaller by one; it also uses a third convention, where the significand m is an integer.

Unit in last place. If $x = \pm 0.d_1 d_2 \dots d_p \cdot \beta^e$ is a floating-point number, we denote by $\mathrm{ulp}(x)$ the weight of the least significand digit of x, i.e., β^{e-p}. (Note that the value of $\mathrm{ulp}(x)$ does not depend on the convention chosen for the (s, m, e) representation.)

Sterbenz's theorem. Sterbenz's theorem says that if x and y are two floating-point numbers of precision p, such that $y/2 \le x \le 2y$, then $x - y$ is exactly representable in precision p. As a consequence, there are no round-off errors when computing $x - y$.

Double-double arithmetic. The "double-double" arithmetic approximates a real number r by the sum of two double-precision numbers, say $x + y$. If x is the rounding-to-nearest of r, and y is the rounding-to-nearest of $r - x$, then double-double arithmetic gives an accuracy which is twice as large as that of a "single" double-precision number.

Other languages (sidebar)

Several other languages than C or C++ provide access to arbitrary precision floating-point arithmetic. For what concerns MPFR, there are interfaces for the Perl, Python, Haskell, Lisp and Ursala languages (see `mpfr.org` for more details). Using MPFR from Fortran is not as easy since one would first have to convert the MPFR data-types to Fortran; however if you want to compute say the exponential of a double-precision number with correct rounding, this is easy, see `http://www.loria.fr/~zimmerma/mpfr/fortran.html`.