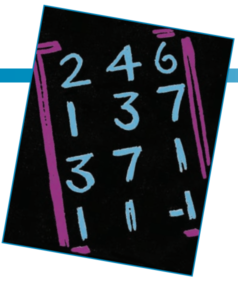
Editors: Konstantin Läufer, laufer@cs.luc.edu
Konrad Hinsen, hinsen@cnrs-orleans.fr



Why and How to Use Arbitrary Precision

By Kaveh R. Ghazi, Vincent Lefèvre, Philippe Théveny, and Paul Zimmermann

Although double precision is usually enough, arbitrary precision increases accuracy and the reproducibility of floating-point computations.

Most floating-point computations today are done in double precision—that is, with a significand (or mantissa, see the "Glossary of Terms" sidebar) of 53 bits. However, some applications require more precision: double-extended precision (64 bits or more), quadruple precision (113 bits), or even more. In a 2001 article in *The Astronomical Journal*, Toshio Fukushima wrote, "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 gave an example in which 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 the electronic control units of aircrafts or nuclear reactors.

As our running example here, we'll assume we want to determine 10 decimal digits of the constant $173746a + 94228b - 78487c$, where $a = \sin(10^{22})$, $b = \log(17.1)$, and $c = \exp(0.42)$. In this simple example, there are no input errors because all values are known exactly—that is, they're known with infinite precision. We ran all experiments reported here with GNU Compiler Collection (GCC) 4.3.2 running on a 64-bit Core 2 under Fedora 10, with GNU C Library 2.9.

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;
}
```

From this program, we get `d = 2.9103830456733704e-11`. This value is completely wrong; 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 (on ARM computers, `long double` is double precision only; on PowerPCs, it corresponds to double-double arithmetic [see the "Glossary of Terms" sidebar], and, under Solaris, it corresponds to quadruple precision). We also change `sin(1e22)` to `sinl(1e22L)`, `log` to `logl`, `exp` to `expl`, and `%.16e` to `%.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.

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 shouldn't 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 Standard" sidebar). However, we can't represent `17.1` exactly in binary, the closest double-precision value being $2406611050876109 \cdot 2^{-47}$, which differs by about $1.4 \cdot 10^{-15}$. The same problem happens with `0.42`.

Second, for a mathematical function, such as sin, and a floating-point input, such as $x = 10^{22}$, the value $\sin x$ usually isn't 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 $6.8 \cdot 10^{-18}$.

Third, as the sidebar notes, IEEE 754 requires neither correct rounding of mathematical functions like sin, log, and exp, nor even some given accuracy, and results are completely platform-dependent.[1] However, while the 1985 version said nothing about these mathematical functions, the 2008 version recommends correct rounding. Thus, the computed values for the variables a, b, and c might differ by several ulps

Glossary of Terms

Radix, Significand, and Exponent

If x is a floating-point number of precision p in radix β, we can write it $x = \pm 0.d_1\, d_2 \ldots d_p \cdot \beta^e$, where $s = \pm 1$ is the sign of x, $m = 0.d_1 d_2 \ldots d_p$ is the significand of x, and e is the exponent of x. This representation is unique if we force d_1 to be non-zero. Also, different conventions are possible for the significand, which lead to different values for the exponent. For example, IEEE 754-2008 uses $m = d_1.d_2 \ldots d_p$, 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 $x = \pm 0.d_1 d_2 \ldots d_p \cdot \beta^e$ is a floating-point number, we denote by ulp(x) the weight of the least significand digit of x, that is, β^{e-p}. (The value of ulp(x) doesn't 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 $y/2 \leq x \leq 2y$, 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

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 that's twice as large as that of a single double-precision number.

The IEEE 754 Standard

IEEE 754 is a widely used standard for floating-point representations and operations that your computer uses everyday.[1] It's very important because it defines floating-point formats, which lets two computers exchange floating-point values without any loss of accuracy. The standard requires correct rounding for arithmetic operations, which guarantees that the same program will yield identical results on two different computers (under some conditions that we omit here). IEEE 754 was first approved in 1985, and was revised in 2008. We describe the revision here, denoted as IEEE 754-2008.

More on IEEE 754-2008

IEEE 754-2008 defines 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` formats 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 GNU Compiler Collection (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 (see www.open-std.org/jtc1/sc22/wg14/www/projects).

Our running example then becomes:

```
#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`. (We had to convert the final result d to `double` because the GNU C library doesn't 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

We can configure the traditional floating-point unit of the 32-bit x86 processors 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, configure their processors so that, by default, they round in double precision. In contrast, under Linux, rounding is done by default in extended precision; this is a bad choice for the reasons detailed elsewhere (www.vinc17.org/research/extended.en.html).

Reference

1. *ANSI-IEEE std. 754-2008, Floating-Point Arithmetic*, IEEE, 2008.

(or units in last place; see the "Glossary of Terms" sidebar) from the correctly rounded result. As we describe later, on this particular platform—whether optimizations are enabled or not—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 $173746a + 94228b - 78487c$.

Other Languages

Several languages other than C or C++ provide access to arbitrary precision floating-point arithmetic. For what concerns Multiple Precision Floating-Point Reliable (MPFR), there are interfaces for the Perl, Python, Haskell, Lisp, and Ursala languages (see www.mpfr.org for more details).

Using MPFR from Fortran is not easy because the MPFR C library uses complex aggregate C data types for representing arbitrary precision numbers. It would be difficult to represent and manipulate these structures from Fortran using the MPFR interface. If, however, you're willing to restrict precision to exactly that of the underlying platform's C data types for double or long double floating point, it's easier to write C wrappers for various MPFR mathematical functions and call these externally from Fortran code (for example, see www.loria.fr/~zimmerma/mpfr/fortran.html).

GNU's Fortran implementation uses MPFR to constant-fold mathematical functions as well. So, if an expression is a compile-time constant formula, the GNU Fortran compiler will replace it with a correctly computed and correctly rounded result, just like the GNU C-family of languages (as we describe in the main text). Constant-folded expressions are of course limited to the precision of the predefined Fortran data types.

Assuming it's 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 of Terms" 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—that is, $2^{-36} \approx 1.5 \cdot 10^{-11}$. That the exact result is of the same order of magnitude explains why our final result d is completely wrong.

The GNU MPFR Library

By *arbitrary precision*, we mean the ability for the user to choose the precision of each calculation. (*Multiple precision* means that large significands are split over several machine words; modern computers can store at most 64 bits—about 20 digits—in one word.) Several programs or libraries let us perform computations in arbitrary precision; this is particularly true for most computer algebra systems such as Mathematica, Maple, or Sage. Here, we focus on GNU Multiple Precision Floating-Point Reliable (MPFR),[2] a C library dedicated to floating-point computations in arbitrary precision (for languages other than C, see the "Other Languages" sidebar).

As the "IEEE 754 Standard" sidebar describes, what makes MPFR different is that it guarantees correct rounding. 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,
        (mpfr_ptr) 0);
      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 `d = -1.3418189578296195e-12` which exactly matches the expected result.

Constant Folding

In a given program, when an expression is a constant, such as $3 + (17 \times 42)$, it can 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 by the compiler is known as *constant folding* (en.wikipedia.org/wiki/constant_folding).

With correctly rounded constant folding, the generated constant depends on the format of the floating-point type on the target platform, and not on the building platform's processor, system, or mathematical library. 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 built-in) mathematical functions such as sin, cos, log, and 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 (that is, using `-O0`), we get as result `2.4957989804940914e-01`. With optimization (that is, 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 that the expression `sin(x)` is a constant, with rounding mode to nearest; it therefore calls MPFR to evaluate it and directly replaces `sin(x)` with its correctly rounded value. (When compiling with `-O1`, we can even omit linking with the mathematical library—that is, `gcc -O1 test.c`—which proves that the mathematical library isn't 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 built-in functions, we can use `gcc -fno-builtin`, or more specifically, `gcc -fno-builtin-sin`, to target the sin function by itself.) The correct value is the one obtained with `-O1`. Note, however, that if the GNU C library doesn't 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 for correct rounding for every input and every function.

Note also that on x86 processors, the GNU C library uses the `fsin` implementation from the x87 coprocessor, which for $x = 0.2522464$ returns the correctly rounded result. However, this is just by chance because among the 10^7 double-precision numbers including 0.25 and above, `fsin` gives an incorrect rounding for 2,452 of them.

As we've shown here, 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 round-off errors, numerical cancellations, errors in binary-decimal conversions, bad numerical quality of mathematical functions, and so on. Using arbitrary precision—such as with the GNU MPFR library—helps to increase the final accuracy. More important, as our constant folding within the GCC example demonstrates, the correct rounding of mathematical functions in MPFR helps increase the reproducibility of floating-point computations among different processors with different compilers and operating systems.

References

1. V. Lefèvre, "Test of Mathematical Functions of the Standard C Library," 2009, www.vinc17.org/research/testlibm.
2. L. Fousse et al., "MPFR: A Multiple-Precision Binary Floating-Point Library With Correct Rounding," *ACM Trans. Mathematical Software*, vol. 33, no. 2, 2007, article 13.

Kaveh R. Ghazi is a software engineer and a Free Software contributor. He is also a member of the GNU Compiler Collection Steering Committee and integrated GCC with the MPFR library to provide the constant folding functionality described here. He has a BS in computer science from Rutgers University. Contact him at ghazi@caip.rutgers.edu.

Vincent Lefèvre is an INRIA researcher at the LIP (Laboratoire de l'Informatique du Parallélisme), École Normale Supérieure de Lyon, France. His research interests include computer arithmetic. Lefèvre received a PhD in computer science from the École Normale Supérieure de Lyon in 2000. Contact him at vincent@vinc17.net.

Philippe Théveny is a software engineer in the INRIA team ARENAIRE at Laboratoire de l'Informatique du Parallélisme in Lyon, France, and a contributor to the MPFR, MPC, and MPFI libraries. Théveny has bachelor degrees in mathematics and computer science from Université de Montpellier, France. Contact him at philippe.theveny@ens-lyon.fr.

Paul Zimmermann is research director at INRIA Nancy–Grand Est, France. His research interests include computational number theory and arbitrary precision arithmetic. He wrote, with Richard Brent, the textbook *Modern Computer Arithmetic* (Cambridge University Press, to appear). Zimmermann has a PhD in computer science from École Polytechnique in Palaiseau, France. He is a member of the program committee of the Symposium on Computer Arithmetic (ARITH). Contact him at Paul.Zimmermann@inria.fr.