Comparison of Floating Point Numbers

Matthias Rupp
Beilstein Endowed Chair for Cheminformatics
Johann Wolfgang Goethe-University
60323 Frankfurt am Main, Germany

July 12, 2007

The following applies to computations using floating point numbers on most platforms and programming languages.

Problem

Direct comparisons of floating point numbers (`x = 0, x < y,` ...) are sources of errors in computer programs and should be avoided. Reasons include

- Excluding singularities but allowing values close to them is pointless.

  `if x != 0.: y = 1./x` avoids division by zero, but allows division by tiny (close to zero) values, leading in turn to huge values which can cause problems later on.

- Precision of previous computations is not considered.

  Intermediate results can be less precise than the machine precision (contain errors): `if x == 0.` will not work if e. g. $x = 10^{-6}$.

- Floating point numbers are not real numbers.

  Limited precision implies approximation, e. g. 0.2 may be stored as `0.200...01`. Floating point operations ($+,-,\cdot,/$) are not associative, commutative or distributive [3]. Errors, e. g. due to rounding, can accumulate. All of this can introduce additional error.

- Results can depend on hardware platform, operating system, compiler and compilation settings.

  Monniaux [4] discusses examples for this. Interesting consequences include that a floating point variable may — under specific circumstances — slightly change its value without an assignment to it.

Solutions

Several solutions are shortly discussed. In the following, x and y are two floating point numbers to be compared; code examples are given in pseudocode, except where mentioned otherwise.

Absolute error bound

The absolute difference $|x - y|$ is computed and compared to a fixed error bound ϵ:

```
float_eq(x,y,epsilon) := abs(x-y) < epsilon
float_lt(x,y,epsilon) := x < y-epsilon
```

Advantages: Fast. Simple. Portable.

Disadvantages: Works only for values in a given range. For example, an error of 1 is great for values about 10^6, but bad for values about 10^{-6}.

Relative error bound

The acceptable error ϵ is specified relative to the size of the involved operands. Let $\operatorname{fexp} x$ denote the exponent of the floating point number x in base 2 representation. Knuth [3] introduces the relationships

$$\begin{aligned} x \prec y &\Leftrightarrow y - x > \epsilon \max\left\{2^{\operatorname{fexp} x}, 2^{\operatorname{fexp} y}\right\} \\ x \sim y &\Leftrightarrow |u - v| \leq \epsilon \max\left(2^{\operatorname{fexp} x}, 2^{\operatorname{fexp} y}\right) \\ x \succ y &\Leftrightarrow x - y > \epsilon \max\left\{2^{\operatorname{fexp} x}, 2^{\operatorname{fexp} y}\right\}. \end{aligned}$$

For given x and y, exactly one of $\{\prec, \sim, \succ\}$ holds. The relationship $\prec$ is transitive and satisfies $x \prec y \Leftrightarrow y \succ x$ as well as $x \prec y \Rightarrow x < y$. A simplified implementation is given by

```
float_eq(x,y,epsilon) := abs(x-y) <= epsilon*max(abs(x),abs(y))
```

Note that one might wish to guard against underflow in the multiplication with ϵ,[1] and consider the special case of numbers very close to zero.[2] There is a public C implementation by Theodore Belding.[3]

Advantages: Portable. Intuitive error specification (as a percentage).

Disadvantages: Slower (involves more operations than the other methods).

[1] `http://www.boost.org/libs/test/doc/components/test_tools/floating_point_comparison.html`
[2] `http://www.cygnus-software.com/papers/comparingfloats/comparingfloats.htm`
[3] `http://fcmp.sourceforge.net/`

Integer interpretation

Most implementations use the IEEE 754 [2] standard. There, two ordered floating point numbers remain ordered when interpreted as sign-magnitude integers. Therefore, floating point numbers can in principle be compared by casting to integer and doing an integer comparison. One can also obtain the next representable floating point number by converting to integer and incrementing. However, when using this for comparison several technical details (two-complement integers, special values such as `inf` and `nan`, subnormal numbers, handling of $+0$ and -0) have to be considered. The following C implementation is due to Bruce Dawson[4]:

```
bool float_eq(float a, float b, int maxulps)
// ulp = units in the last place; maxulps = maximum number of
// representable floating point numbers by which x and y may differ.
{
  // maxulps small enough so NaN will not be equal to other numbers.
  assert(maxulps > 0 && maxulps < 4 * 1024 * 1024);

  // convert to integer.
  int aint = *(int*)&a;
  int bint = *(int*)&b;

  // make lexicographically ordered as a twos-complement int
  if (aint < 0) aint = 0x80000000 - aint;
  if (bint < 0) bInt = 0x80000000 - bint;

  // compare.
  return abs(aint - bint) <= maxulps;
}
```

Advantages: Fast (on machines where floating point and integer operations can be done in the same registers, i. e. without going through memory). Error can be specified in ulps (units in the last place).

Disadvantages: Less portable as it depends on hardware specifications (IEEE 754 compliance, integers with the same size as the floating point numbers must be available).

References

[1] David Goldberg. What every computer scientist should know about floating point arithmetic. *ACM Computing Surveys*, 23(1):5–48, 1991.

[2] IEEE. IEEE standard for binary floating-point arithmetic (IEEE standard 754-1985), 1985.

[3] Donald Knuth. *Seminumerical Algorithms*, volume 2 of *The Art Of Computer Programming*. Addison-Wesley, 1997.

[4] David Monniaux. The pitfalls of verifying floating-point computations. *ACM Transactions on Programming Languages and Systems*, to appear, 2007.

[4] `http://www.cygnus-software.com/papers/comparingfloats/comparingfloats.htm`