Epsilon Equal

Roy Ratcliffe[1,*]

Abstract

The concept of equality for floating-point numbers in the C language is not straightforward due to the limitations of floating-point arithmetic precision. This paper explores the intricacies of determining equality between floating-point numbers and discusses the implications of these differences, particularly in scenarios involving real-world computations. The use of a pure logic solution is presented, along with its implementation and considerations. Additionally, the paper introduces a C99 solution optimized for embedded systems, offering a practical approach to address the challenges of comparing floating-point numbers. The paper concludes by advocating for the use of ϵ-equality, a method that leverages factors of the smallest possible difference to determine equality, and highlights its relevance in the context of embedded systems with a floating-point unit.

Keywords:
C

In C, when are two floating-point numbers equal? The answer is not exactly simple. It depends on what equal means. One might assume naively that `x == y` if answers *true* the number at x and y match, but since floating-point arithmetic attempts to model real numbers within a limited level of precision.

Most of the time, the distinction is not significant. In the real world, the difference sometimes **does** matter, however, e.g. where values derive from computations. C compilers may even issue a warning message when comparing floats.

```
Rcpp::evalCpp("0.0 == 0.0")
```

```
[1] TRUE
```

```
Rcpp::evalCpp("0.0 == DBL_EPSILON")
```

```
[1] FALSE
```

This result demonstrates that two quantities need only differ by a minuscule amount for inequality when under direct comparison. The standard C math library defines `DBL_EPSILON` as the smallest possible floating-point positive number.

1. Solution in Pure logic

In logic, the solution appears below.

```
%!  epsilon_equal(+X:number, +Y:number) is semidet.
%!  epsilon_equal(+Epsilons:number, +X:number, +Y:number) is semidet.
%
%   Succeeds only when the absolute  difference   between  the two given
```

[*]Corresponding author
Email address: roy@ratcliffe.me (Roy Ratcliffe)
[1]See more at GitHub.

```
%   numbers X and Y is less than  or   equal  to epsilon, or some factor
%   (Epsilons) of epsilon according to rounding limitations.

epsilon_equal(X, Y) :- epsilon_equal(1, X, Y).

epsilon_equal(Epsilons, X, Y) :- Epsilons * epsilon >= abs(X - Y).
```

2. In C99

The C99 solution works better for embedded systems where fancy backtracking logic is not readily available.

```
#pragma once

/*
 * float.h for DBL_EPSILON and friends
 * math.h for fabs(3) and friends
 */
#include <float.h>
#include <math.h>

/*!
 * \brief Epsilon equality for double-precision floating-point numbers.
 * \details Succeeds only when the absolute difference between the two given
 * numbers X and Y is less than or equal to epsilon, or some factor (epsilons)
 * of epsilon according to rounding limitations.
 */
// [[Rcpp::export]]
static inline bool fepsiloneq(unsigned n, double x, double y) {
  return n * DBL_EPSILON >= fabs(x - y);
}

/*!
 * \brief Epsilon equality for single-precision floating-point numbers.
 */
// [[Rcpp::export]]
static inline bool fepsiloneqf(unsigned n, float x, float y) {
  return n * FLT_EPSILON >= fabsf(x - y);
}
```

The ϵ-equal family use factors of the smallest possible difference to determine equality. Two function implementations exist: one for single- and another for double-precision.

The implementation avoids division since that reduces precision. Subtraction computes the differences between two measurements. The $n \times \epsilon$ threshold typically computes out at compile time provided that `n` is some constant factor—since the functions appear as `static` and `inline`. That makes the computation of equality pretty light on the floating-point unit.

Now we can apply an ϵ-precision level when matching real numbers. In the examples below, the second test fails because 1×10^{-15} exceeds the ϵ equality threshold. It would succeed if using `fepsiloneqf` because single-precision floating-point ϵ is a larger quantity.

```
fepsiloneq(1, 0.0, 0.0)
```

```
[1] TRUE
```

```r
fepsiloneq(1, 0.0, 1e-15)
```

```
[1] FALSE
```

```r
fepsiloneq(1, 0.0, 1e-16)
```

```
[1] TRUE
```

3. Conclusions

For embedded systems, this approach assumes a floating-point unit, else a soft-float library. Embedded FPUs are not uncommon these days, especially in 32-bit cores.

In effect, ϵ-equality amounts to

$$x - n\epsilon \leq \ y \leq x + n\epsilon$$

or y between $x \pm n\epsilon$ or vice versa. Such comparisons become, effectively, a symmetrical interval intersection test. Ideal for floating-point number matching.