

Comparison of Floating-point Values

Yong-Ming Li

March, 2009

1 Overview

As a rule of thumb in programming, we should avoid direct comparison of two floating-point values for equality because computer has limited precision. It is widely regarded that direct comparisons of floating-point values are the sources of instability that can be triggered by arithmetic computations, change of compilers, or even change of compiling options. Some people think that a solution to such anomalies is simply not to compare floating-point numbers for equality but instead to consider them equal if they are within some error bound ε . This is hardly a cure all, because it raises as many questions as it answers. What should the value of ε be? If $x < 0$ and $y > 0$ are within ε , should they really be considered equal, even though they have different signs? If one does a Google search, he would find many research papers on this topic. A few relevant publications can be found in the references section.

In the subsequent sections, I will discuss the so called machine precision and several comparison methods.

2 Machine precision

In floating-point arithmetic, the machine precision (also called machine tolerance) is, for a particular floating-point unit, the difference between 1 and the smallest exactly representable number greater than one. In other words, if we add 1 by a small number that is larger than the machine precision would result in a number larger than 1. The exact value of machine precision depends on the particular floating-point format used, such as float, double, long double, or similar as supported by the programming language, the compiler, and the runtime library for the actual platform. In practice, we can use the following C program to determine the machine precision under certain environment.

```
double epsilon = 1.0;

while ((1.0 + epsilon) != 1.0)
{
    epsilon /= 2.0;
}
printf( "\nCalculated Machine epsilon: %.15e\n\n", epsilon );
```

Running the above program under Microsoft Visual C++ 2008 compiler, we would get $\varepsilon = 1.110223024625157 \times 10^{-16}$. It should be pointed out that this ε is a half of Microsoft published

machine precision (Referring to float.h). If we look back the code, we notice that ε is repeatedly reduced by factor of 2. Therefore, Microsoft must have taken the upper bound while I took the lower bound. A simple program test, as illustrated below, would reveal that the lower bound is safer:

```

if ( (1.0 + 0.5 * epsilon_microsoft) == 1.0)
    Yes, it will come here.
end if

if ( (1.0 + 0.50001 * epsilon_microsoft) == 1.0)
    Note epsilon is the smallest number such that 1.0+epsilon != 1.0.
    By reducing epsilon almost by half we would expect it to come
    here but it will not.
end if

```

3 Comparison with absolute error bound

The absolute difference $|a - b|$ is computed and compared to a fixed error bound, say the machine tolerance ε :

$$|a - b| < \varepsilon$$

If such condition is true, we consider a is equal to b . The advantages of comparison with absolute are fast, simple and portable. Some severe disadvantage includes no consideration of the size of a number. For example, it is much more difficult to reach 1.0×10^{-6} accuracy for a very large number than a very small number. With absolute error bound, both cases will be treated equally. Furthermore, it pays no attention to differences of *significant digits*. Assume $a = 2.2204460 \times 1.0^{-16}$ and $b = 1.110223024625157 \times 10^{-16}$. By using the absolute error bound, we would consider them to be equal even though their *relative error* ($\frac{|a-b|}{a}$) is 0.5

The significant digits of a number are those digits that carry meaning contributing to its accuracy. This includes all digits except:

1. leading and trailing zeros (unless a decimal point is present) where they serve merely as placeholders to indicate the scale of the number.
2. spurious digits introduced, for example, by calculations carried out to greater accuracy than that of the original data, or measurements reported to a greater precision than the equipment supports.

Scientists, mathematicians and engineers prefer representing a number in normalized scientific notation such as

$$a \times 10^b$$

with a being less than 10 but equal or greater than 1. Since placeholder leading and trailing zeros do not occur in normalized scientific notation, so all digits of a are significant.

4 Comparison with relative error bound

In many applications, we are interested in knowing the differences of significant digits rather than the absolute difference of two floating-point numbers. In this case, the following comparison makes

more sense than the absolute error bound:

$$|a - b| < \frac{\max(|a|, |b|)}{10^{15}}$$

If such condition is true, we say a and b are equal within 15 digits. In some publications, the following formula is used:

$$|a - b| < \max(|a|, |b|) \times \varepsilon$$

If either $|a|$ or $|b|$ is not a zero, we can write the above relation as:

$$\frac{|a - b|}{\max(|a|, |b|)} < \varepsilon.$$

As it is seen, this is a relative error bound. In other words, the error is measured as the percentage of the size of the larger number. In general, the relative error makes more sense than the absolute error since it takes into consideration the size of the number. However, users need to pay special attentions to the following aspects:

1. When both a and b are very small, it may result in underflow without safeguard in division by 10^{15} or multiplication by ε .
2. From a practical point of view, we would treat $a = 1.0 \times 10^{-30}$ as zero. However, it will not be considered equal to zero using the relative error bound since they do not have any significant digit to be equal.

5 Geometric comparison

In CAD system, we work usually with geometric entities. Therefore, parameter, ratio, poles, and so on have geometric meaning. Accordingly, we should use the well-defined tolerance such knot tolerance, angular tolerance, and distance tolerance for comparing geometric related floating point values. Sometime, these values are not directly related to our well-defined tolerance. But it may be indirectly related to it - We just need some deep analysis. For example, the `m_minorMajorRatio` of an ellipse is defined as $\frac{b}{a}$ where a and b are respectively the length of major and minor axis of the ellipse. Although `m_minorMajorRatio` is a unitless scalar, it is related to the lengths of major and minor axes. Therefore, such comparison has geometric meaning:

```
if (fabs(m_minorMajorRatio - 1.0) < dist_tol / MAX(a, b)) then
    Ellipse can be considered as circle.
    The deviation will be less than the distance tolerance.
end if
```

For efficiency, math uses the following comparison:

```
if (fabs(m_minorMajorRatio - 1.0) < GetRelativeTolerance()) then
    Ellipse can be considered as circle.
end if
```

As far as the length of major (or minor) axis is less than 1.0×10^4 meter, the above comparison is stricter than the first one.

6 How to fix problems

Based on my experience in fixing direct comparison of doubles in our code, I recommend:

- If any or both of a and b have participated in arithmetic operations, it is not safe for direct comparison for equality. We need to change the code to use an error bound comparison.
- If a number is not representable in binary floating-point of any finite precision, it is better to use an error bound comparison. Otherwise, changes of compilers or compiler options may result in a bad surprising. A simple example of non-representable number is 0.1 as the exact binary representation of 0.1 would have an 1100 sequence continuing endlessly as:

$$e = -??; s = 1100110011001100110011001100110011\dots,$$

where, s is the significand and e is the exponent. When rounded to 24 bits this becomes

$$e = -27; s = 110011001100110011001101,$$

which is actually 0.100000001490116119384765625 in decimal.

- Although 0.0 is representable in binary floating-point of any finite precision, the following usage or similar should be avoided:

```
if (a != 0.0)
    c = b / a;
end if
```

This is because division by a very small number is numerically instable. Furthermore, it may result in an overflow exception.

Since the floating-point arithmetic is more computer science and architecture than math, other experts' suggestions and comments are certainly more than welcomed.

7 Round number to given decimal place

It is often desired to round a number to a specified decimal place. Assume we want to round 0.084849999 to the third decimal place to obtain 0.085, we would

- Multiply the number by 1000 to get 84.849999.
- Add 0.5 to round to the nearest integer. The result would be 85.349999.
- Truncate all decimals. The result is 85.0.
- Finally, divide the truncated number by 1000.

By generalizing the above steps, we have the following implementation:

```

void IMRoundValue(
    double  dValue,      // Input: the number to be rounded
    int     nDecimals,  // Input: the decimal points you want to keep
    double  *dRndValue) // Output: the round-off number
{
    double  temp, dScale;

    // Scale the number and add 0.5 such that the 1st decimal of enlarged number
    // will be round to the nearest integer if it is equal or larger than 0.5.
    dScale = pow(10.0, nDecimals);
    temp = dScale * dValue + 0.5;

    // Truncate the decimal numbers and scale it back
    temp = floor(temp);
    *dRndValue = temp / dScale;
}

```

As it was discussed previously, computer has only limited precision. Therefore, some numbers just cannot be presented exactly. We take the same number (0.084849999) as an example. Assume we want to round it to the fifth decimal place. The expected result is 0.08485. Instead, we will get 0.084849999999999995 when running Microsoft Visual C++ 2008.

8 Conclusions

In this document, we discussed several methods used for comparing floating-point numbers. It should be pointed out that there are other comparison methods as well (e.g., comparing binary floats). From our discussion we can see that it is not trivial at all to select a right comparison method. My recommendation is that we should always choose geometric comparison if we are dealing with geometric property. When we use other two methods in comparing two floating-point numbers, we need to be aware of their advantages and disadvantages.

References

- [1] *What Every Computer Scientist Should Know About Floating-Point Arithmetic*, by David Goldberg, published in the March, 1991 issue of Computing Surveys.
- [2] *The Art of Computer Programming*, by Donald Knuth, Volume 2: Seminumerical Algorithms, Third Edition. Addison-Wesley, 1997. ISBN 0-201-89684-2. Section 4.2: floating-point Arithmetic, pp.214-264.
- [3] *How To Work Around Floating-Point Accuracy/Comparison Problems*, published in Microsoft support website (<http://support.microsoft.com/kb/69333>).
- [4] *Standard for Binary Floating-Point Arithmetic*, IEEE 754.