How do you compare float and double while accounting for precision loss?

Asked 16 years, 4 months ago Modified 9 months ago Viewed 675k times

673

What would be the most efficient way to compare two `double` or two `float` values?

Simply doing this is not correct:

```
bool CompareDoubles1 (double A, double B)
{
   return A == B;
}
```

But something like:

```
bool CompareDoubles2 (double A, double B)
{
   diff = A - B;
   return (diff < EPSILON) && (-diff < EPSILON);
}
```

Seems to waste processing.

Does anyone know a smarter float comparer?

c++ algorithm optimization floating-point

Share

Improve this question

Follow

edited Sep 18, 2023 at 6:20

Jan Schultke

38.3k ● 8 ● 87 ● 168

asked Aug 20, 2008 at 2:09

Alex

6,771 ● 3 ● 19 ● 3

3 > would it be more efficient to add ... in the beginning of the function? `<invoke Knuth>` Premature optimization is the root of all evil. `</invoke Knuth>` Just go with abs(a-b) < EPS as noted above, it's clear and easy to understand. – Andrew Coleson Aug 20, 2008 at 5:55

3 Here it is the way implemented in Boost Test Library: http://www.boost.org/doc/libs/1_36_0/libs/test/doc/html/utf/testing-tools/floating_point_comparison.html – Alessandro Jacopson Oct 31, 2008 at 13:38

3 The only thing unoptimal about original poster's implementation is that it contains an extra branch at &&. OJ's answer is optimal. fabs is an intrinsic which is a single instruction on x87,

and i suppose on almost anything else too. Accept OJ's answer already! – 3yE Mar 27, 2010 at 17:04

4 If you can, drop the floating point and use fixed points. Example, use {fixed point} millimeters instead of {floating point} meters. – Thomas Matthews Jun 11, 2012 at 2:11

51 *"Simply doing this is not correct"* - This is mere rubbish, of course using `==` can be perfectly correct, but this entirely depends on the context not given in the question. Until that context is known, `==` still stays the *"most efficient way"*. – Christian Rau May 13, 2013 at 7:39

32 Answers

Sorted by: Highest score (default)

569

Be extremely careful using any of the other suggestions. It all depends on context.

I have spent a long time tracing bugs in a system that presumed `a==b` if `|a-b|<epsilon`. The underlying problems were:

1. The implicit presumption in an algorithm that if `a==b` and `b==c` then `a==c`.
2. Using the same epsilon for lines measured in inches and lines measured in mils (.001 inch). That is `a==b` but `1000a!=1000b`. (This is why `AlmostEqual2sComplement` asks for the epsilon or max ULPS).
3. The use of the same epsilon for both the cosine of angles and the length of lines!
4. Using such a compare function to sort items in a collection. (In this case using the builtin C++ operator `==` for doubles produced correct results.)

Like I said: it all depends on context and the expected size of `a` and `b`.

By the way, `std::numeric_limits<double>::epsilon()` is the "machine epsilon". It is the difference between `1.0` and the next value representable by a double. I guess that it could be used in the compare function but only if the expected values are less than 1. (This is in response to @cdv's answer...)

Also, if you basically have `int` arithmetic in `doubles` (here we use doubles to hold int values in certain cases) your arithmetic will be correct. For example `4.0/2.0` will be the same as `1.0+1.0`. This is as long as you do not do things that result in fractions (`4.0/3.0`) or do not go outside of the size of an int.

Share

Improve this answer

Follow

edited May 1, 2022 at 0:17

user17242583

answered Sep 16, 2008 at 22:06

Andrew Stein

13.1k ● 5 ● 38 ● 44

21 +1 for pointing out the obvious (that often gets ignored). For a generic method, you can make the epsilon relative to `fabs(a)+fabs(b)` but with compensating for NaN, 0 sum and

overflow, this gets quite complex. – peterchen Aug 7, 2010 at 6:41

4 There must be something I don't understand. The typical `float` / `double` is **MANTISSA** x 2^ **EXP**. `epsilon` will be dependent on the exponent. For example if the *mantissa* is 24bits and the *exponent* is signed 8bit, then `1/(2^24)*2^127` or `~2^103` is an `epsilon` for some values; or is this referring to a minimum *epsilon*? – artless-noise-bye-due2AI Mar 17, 2013 at 18:54

4 Wait a second. Is what I said what you meant? You are saying why `|a-b|<epsilon`, is *not* correct. Please add this link to your answer; if you agree cygnus-software.com/papers/comparingfloats/comparingfloats.htm and I can remove my dumb comments. – artless-noise-bye-due2AI Mar 17, 2013 at 19:16

3 +1 Which arise the need for using a special error margin based on scenario and what are you comparing – Khaled.K May 28, 2013 at 9:04

9 This is a very long comment, not an answer in itself. Is there a (set of) canonical answer(s) for all contexts? – Merlyn Morgan-Graham Jul 30, 2016 at 2:49

210

The comparison with an epsilon value is what most people do (even in game programming).

You should change your implementation a little though:

```
bool AreSame(double a, double b)
{
    return fabs(a - b) < EPSILON;
}
```

Edit: Christer has added a stack of great info on this topic on a recent blog post. Enjoy.

Share

Improve this answer

Follow

edited Nov 20, 2015 at 2:59

Trevor Hickey

37.7k ● 35 ● 177 ● 283

answered Aug 20, 2008 at 2:14

OJ.

29.4k ● 6 ● 58 ● 71

13 @DonReba: Only if `EPSILON` is defined as `DBL_EPSILON`. Normally it will be a specific value chosen depending on the required accuracy of the comparison. – Nemo157 Dec 20, 2011 at 20:38

11 `EPSILON` comparison does not work when the floats are large, as the difference between consecutive floats also becomes large. See this article. – kevintodisco Oct 23, 2013 at 4:04

35 No wonder there is Z-fighting in some games when textures/objects far away flicker, like in Battlefield 4. Comparing the difference with `EPSILON` is pretty much useless. You need to compare with a threshold that makes sense for the units at hand. Also, use `std::abs` since it is overloaded for different floating point types. – Maxim Egorushkin Feb 19, 2014 at 16:40

33 I downvoted since the example code shows the typical bug whis is repeated by the majority of programmers. Floating point is always about relative errors, since it is floating point (not fixed point). So it will never correctly work with a fixed error (epsilon). – user2261015 Apr 13, 2017 at 7:58

2 @SirGuy, please post an answer to demonstrate how to properly do this then, or link to one here. I'd like to see non-epsilon-based alternatives. – Gabriel Staples Nov 26, 2020 at 2:36

157

Comparing floating point numbers for depends on the context. Since even changing the order of operations can produce different results, it is important to know how "equal" you want the numbers to be.

Comparing floating point numbers by Bruce Dawson is a good place to start when looking at floating point comparison.

The following definitions are from The art of computer programming by Knuth:

```
bool approximatelyEqual(float a, float b, float epsilon)
{
    return fabs(a - b) <= ( (fabs(a) < fabs(b) ? fabs(b) : fabs(a)) * epsilon);
}

bool essentiallyEqual(float a, float b, float epsilon)
{
    return fabs(a - b) <= ( (fabs(a) > fabs(b) ? fabs(b) : fabs(a)) * epsilon);
}

bool definitelyGreaterThan(float a, float b, float epsilon)
{
    return (a - b) > ( (fabs(a) < fabs(b) ? fabs(b) : fabs(a)) * epsilon);
}

bool definitelyLessThan(float a, float b, float epsilon)
{
    return (b - a) > ( (fabs(a) < fabs(b) ? fabs(b) : fabs(a)) * epsilon);
}
```

Of course, choosing epsilon depends on the context, and determines how equal you want the numbers to be.

Another method of comparing floating point numbers is to look at the ULP (units in last place) of the numbers. While not dealing specifically with comparisons, the paper What every computer scientist should know about floating point numbers is a good

resource for understanding how floating point works and what the pitfalls are, including what ULP is.

Share

Improve this answer

Follow

edited Dec 15, 2016 at 16:13

Damian

4,611 ● 4 ● 43 ● 69

answered Oct 31, 2008 at 15:13

mch

7,363 ● 3 ● 33 ● 34

1 `fabs(a - b) <= ( (fabs(a) < fabs(b) ? fabs(b) : fabs(a)) * epsilon);` saved my live. LOL Note that this version (I haven't checked if is applies for the others too) also considers the change that might occur in the integral part of the floating point number (example: `2147352577.9999997616 == 2147352576.0000000000` where you can clearly see that there is almost a difference of `2` between the two numbers) which is quite nice! This happens when the accumulated rounding error overflows the decimal part of the number. – rbaleksandar Oct 12, 2016 at 12:24

Very nice and helpful article by Bruce Dawson, thanks! – BobMorane Aug 28, 2018 at 19:14

9 Given that this question is tagged C++, your checks would be easier to read being written as `std::max(std::abs(a), std::abs(b))` (or with `std::min()`); `std::abs` in C++ is overloaded with float & double types, so it works just fine (you can always keep `fabs` for readability though). – Razakhel Oct 1, 2018 at 17:09

`definitelyGreaterThan` is reporting **true** for something that should definitely be equal to, i.e. **not** greater than. – mwpowellhtx Jan 19, 2019 at 21:05

2 Turns out the problem was in my code, difference between the original expected value and the parsed string. – mwpowellhtx Jan 20, 2019 at 17:31

124

I found that the Google C++ Testing Framework contains a nice cross-platform template-based implementation of AlmostEqual2sComplement which works on both doubles and floats. Given that it is released under the BSD license, using it in your own code should be no problem, as long as you retain the license. I extracted the below code from ~~http://code.google.com/p/googletest/source/browse/trunk/include/gtest/internal/gtest-internal.h~~ https://github.com/google/googletest/blob/master/googletest/include/gtest/internal/gtest-internal.h and added the license on top.

Be sure to #define GTEST_OS_WINDOWS to some value (or to change the code where it's used to something that fits your codebase - it's BSD licensed after all).

Usage example:

```
double left  = // something
double right = // something
const FloatingPoint<double> lhs(left), rhs(right);
```

```
if (lhs.AlmostEquals(rhs)) {
  //they're equal!
}
```

Here's the code:

```
// Copyright 2005, Google Inc.
// All rights reserved.
//
// Redistribution and use in source and binary forms, with or without
// modification, are permitted provided that the following conditions are
// met:
//
//     * Redistributions of source code must retain the above copyright
// notice, this list of conditions and the following disclaimer.
//     * Redistributions in binary form must reproduce the above
// copyright notice, this list of conditions and the following disclaimer
// in the documentation and/or other materials provided with the
// distribution.
//     * Neither the name of Google Inc. nor the names of its
// contributors may be used to endorse or promote products derived from
// this software without specific prior written permission.
//
// THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
// "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
// LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
// A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
// OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
// SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
// LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
// DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
// THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
// (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
// OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
//
// Authors: wan@google.com (Zhanyong Wan), eefacm@gmail.com (Sean Mcafee)
//
// The Google C++ Testing Framework (Google Test)


// This template class serves as a compile-time function from size to
// type.  It maps a size in bytes to a primitive type with that
// size. e.g.
//
//   TypeWithSize<4>::UInt
//
// is typedef-ed to be unsigned int (unsigned integer made up of 4
// bytes).
//
// Such functionality should belong to STL, but I cannot find it
// there.
//
// Google Test uses this class in the implementation of floating-point
// comparison.
//
// For now it only handles UInt (unsigned int) as that's all Google Test
// needs.  Other types can be easily added in the future if need
// arises.
template <size_t size>
```

```
class TypeWithSize {
 public:
  // This prevents the user from using TypeWithSize<N> with incorrect
  // values of N.
  typedef void UInt;
};

// The specialization for size 4.
template <>
class TypeWithSize<4> {
 public:
  // unsigned int has size 4 in both gcc and MSVC.
  //
  // As base/basictypes.h doesn't compile on Windows, we cannot use
  // uint32, uint64, and etc here.
  typedef int Int;
  typedef unsigned int UInt;
};

// The specialization for size 8.
template <>
class TypeWithSize<8> {
 public:
#if GTEST_OS_WINDOWS
  typedef __int64 Int;
  typedef unsigned __int64 UInt;
#else
  typedef long long Int;  // NOLINT
  typedef unsigned long long UInt;  // NOLINT
#endif  // GTEST_OS_WINDOWS
};


// This template class represents an IEEE floating-point number
// (either single-precision or double-precision, depending on the
// template parameters).
//
// The purpose of this class is to do more sophisticated number
// comparison.  (Due to round-off error, etc, it's very unlikely that
// two floating-points will be equal exactly.  Hence a naive
// comparison by the == operation often doesn't work.)
//
// Format of IEEE floating-point:
//
//   The most-significant bit being the leftmost, an IEEE
//   floating-point looks like
//
//     sign_bit exponent_bits fraction_bits
//
//   Here, sign_bit is a single bit that designates the sign of the
//   number.
//
//   For float, there are 8 exponent bits and 23 fraction bits.
//
//   For double, there are 11 exponent bits and 52 fraction bits.
//
//   More details can be found at
//   http://en.wikipedia.org/wiki/IEEE_floating-point_standard.
//
// Template parameter:
//
//   RawType: the raw floating-point type (either float or double)
```

```
template <typename RawType>
class FloatingPoint {
 public:
  // Defines the unsigned integer type that has the same size as the
  // floating point number.
  typedef typename TypeWithSize<sizeof(RawType)>::UInt Bits;

  // Constants.

  // # of bits in a number.
  static const size_t kBitCount = 8*sizeof(RawType);

  // # of fraction bits in a number.
  static const size_t kFractionBitCount =
    std::numeric_limits<RawType>::digits - 1;

  // # of exponent bits in a number.
  static const size_t kExponentBitCount = kBitCount - 1 - kFractionBitCount;

  // The mask for the sign bit.
  static const Bits kSignBitMask = static_cast<Bits>(1) << (kBitCount - 1);

  // The mask for the fraction bits.
  static const Bits kFractionBitMask =
    ~static_cast<Bits>(0) >> (kExponentBitCount + 1);

  // The mask for the exponent bits.
  static const Bits kExponentBitMask = ~(kSignBitMask | kFractionBitMask);

  // How many ULP's (Units in the Last Place) we want to tolerate when
  // comparing two numbers.  The larger the value, the more error we
  // allow.  A 0 value means that two numbers must be exactly the same
  // to be considered equal.
  //
  // The maximum error of a single floating-point operation is 0.5
  // units in the last place.  On Intel CPU's, all floating-point
  // calculations are done with 80-bit precision, while double has 64
  // bits.  Therefore, 4 should be enough for ordinary use.
  //
  // See the following article for more details on ULP:
  // http://www.cygnus-software.com/papers/comparingfloats/comparingfloats.htm.
  static const size_t kMaxUlps = 4;

  // Constructs a FloatingPoint from a raw floating-point number.
  //
  // On an Intel CPU, passing a non-normalized NAN (Not a Number)
  // around may change its bits, although the new value is guaranteed
  // to be also a NAN.  Therefore, don't expect this constructor to
  // preserve the bits in x when x is a NAN.
  explicit FloatingPoint(const RawType& x) { u_.value_ = x; }

  // Static methods

  // Reinterprets a bit pattern as a floating-point number.
  //
  // This function is needed to test the AlmostEquals() method.
  static RawType ReinterpretBits(const Bits bits) {
    FloatingPoint fp(0);
    fp.u_.bits_ = bits;
    return fp.u_.value_;
  }
```

```
  // Returns the floating-point number that represent positive infinity.
  static RawType Infinity() {
    return ReinterpretBits(kExponentBitMask);
  }

  // Non-static methods

  // Returns the bits that represents this number.
  const Bits &bits() const { return u_.bits_; }

  // Returns the exponent bits of this number.
  Bits exponent_bits() const { return kExponentBitMask & u_.bits_; }

  // Returns the fraction bits of this number.
  Bits fraction_bits() const { return kFractionBitMask & u_.bits_; }

  // Returns the sign bit of this number.
  Bits sign_bit() const { return kSignBitMask & u_.bits_; }

  // Returns true iff this is NAN (not a number).
  bool is_nan() const {
    // It's a NAN if the exponent bits are all ones and the fraction
    // bits are not entirely zeros.
    return (exponent_bits() == kExponentBitMask) && (fraction_bits() != 0);
  }

  // Returns true iff this number is at most kMaxUlps ULP's away from
  // rhs.  In particular, this function:
  //
  //   - returns false if either number is (or both are) NAN.
  //   - treats really large numbers as almost equal to infinity.
  //   - thinks +0.0 and -0.0 are 0 DLP's apart.
  bool AlmostEquals(const FloatingPoint& rhs) const {
    // The IEEE standard says that any comparison operation involving
    // a NAN must return false.
    if (is_nan() || rhs.is_nan()) return false;

    return DistanceBetweenSignAndMagnitudeNumbers(u_.bits_, rhs.u_.bits_)
        <= kMaxUlps;
  }

 private:
  // The data type used to store the actual floating-point number.
  union FloatingPointUnion {
    RawType value_;  // The raw floating-point number.
    Bits bits_;      // The bits that represent the number.
  };

  // Converts an integer from the sign-and-magnitude representation to
  // the biased representation.  More precisely, let N be 2 to the
  // power of (kBitCount - 1), an integer x is represented by the
  // unsigned number x + N.
  //
  // For instance,
  //
  //   -N + 1 (the most negative number representable using
  //          sign-and-magnitude) is represented by 1;
  //   0      is represented by N; and
  //   N - 1  (the biggest number representable using
  //          sign-and-magnitude) is represented by 2N - 1.
  //
  // Read http://en.wikipedia.org/wiki/Signed_number_representations
```

```
  // for more details on signed number representations.
  static Bits SignAndMagnitudeToBiased(const Bits &sam) {
    if (kSignBitMask & sam) {
      // sam represents a negative number.
      return ~sam + 1;
    } else {
      // sam represents a positive number.
      return kSignBitMask | sam;
    }
  }

  // Given two numbers in the sign-and-magnitude representation,
  // returns the distance between them as an unsigned number.
  static Bits DistanceBetweenSignAndMagnitudeNumbers(const Bits &sam1,
                                                     const Bits &sam2) {
    const Bits biased1 = SignAndMagnitudeToBiased(sam1);
    const Bits biased2 = SignAndMagnitudeToBiased(sam2);
    return (biased1 >= biased2) ? (biased1 - biased2) : (biased2 - biased1);
  }

  FloatingPointUnion u_;
};
```

EDIT: This post is 4 years old. It's probably still valid, and the code is nice, but some people found improvements. Best go get the latest version of `AlmostEquals` right from the Google Test source code, and not the one I pasted up here.

Share

Improve this answer

Follow

edited Jul 4, 2018 at 22:13

Bo R

2,343 ● 1 ● 11 ● 18

answered Aug 6, 2010 at 11:24

skrebbel

9,931 ● 8 ● 36 ● 36

3 +1: I agree this one is correct. However, it doesn't explain why. See here: cygnus-software.com/papers/comparingfloats/comparingfloats.htm I read this blog post after I wrote my comment on the top score here; I believe it says the same thing and provides the rational/solution that is implemented above. Because there is so much code, people will miss the answer. – artless-noise-bye-due2AI Mar 17, 2013 at 19:07

There are a couple of nasty things that can happen when implicit casts occur doing say FloatPoint<double> fp(0.03f). I made a couple modifications to this to help prevent that. template<typename U> explicit FloatingPoint(const U& x) { if(typeid(U).name() != typeid(RawType).name()) { std::cerr << "You're doing an implicit conversion with FloatingPoint, Don't" << std::endl; assert(typeid(U).name() == typeid(RawType).name()); } u_.value_ = x; } – JeffCharter Mar 17, 2014 at 21:16

2 Good find! I guess it would be best to contribute them to Google Test, though, where this code was stolen from. I'll update the post to reflect that probably there's a newer version. If the Google guys act itchy, could you put it in e.g. a GitHub gist? I'll link to that as well, then. – skrebbel May 1, 2014 at 12:48

Am I the only one getting 'false' by comparing the doubles 0 with 1e-16? 0's biased representation is 9223372036854775808 while 1e-16's biased representation is 13590969439990876604. It seems to be a discontinuity in the representation or am I doing something wrong? – Gaston Oct 1, 2014 at 17:17

3 For the newest code snippet, see here and here. – Jaege Nov 25, 2016 at 3:11

49

For a more in depth approach read Comparing floating point numbers. Here is the code snippet from that link:

```
// Usable AlmostEqual function
bool AlmostEqual2sComplement(float A, float B, int maxUlps)
{
    // Make sure maxUlps is non-negative and small enough that the
    // default NAN won't compare as equal to anything.
    assert(maxUlps > 0 && maxUlps < 4 * 1024 * 1024);
    int aInt = *(int*)&A;
    // Make aInt lexicographically ordered as a twos-complement int
    if (aInt < 0)
        aInt = 0x80000000 - aInt;
    // Make bInt lexicographically ordered as a twos-complement int
    int bInt = *(int*)&B;
    if (bInt < 0)
        bInt = 0x80000000 - bInt;
    int intDiff = abs(aInt - bInt);
    if (intDiff <= maxUlps)
        return true;
    return false;
}
```

Share Improve this answer Follow

answered Aug 20, 2008 at 6:31

grom
16.1k ● 19 ● 65 ● 67

16 What is the suggested value of maxUlps? – unj2 Aug 1, 2011 at 1:10

6 Will " `*(int*)&A;` " violate strict aliasing rule? – osgx Aug 11, 2011 at 5:31

3 According to gtest (search for ULP), 4 is an acceptable number. – May Oakes Jul 17, 2012 at 20:30

4 And here are a couple updates to Bruce Dawson's paper (one of which is linked in the paper's intro): randomascii.wordpress.com/2012/02/25/… and randomascii.wordpress.com/2012/06/26/… – Michael Burr Aug 14, 2012 at 18:07

6 It took me a awhile to figure out what on ULP was: Units in the Last Place – JeffCharter Mar 13, 2014 at 16:19

40

Realizing this is an old thread but this article is one of the most straight forward ones I have found on comparing floating point numbers and if you want to explore more it has more detailed references as well and it the main site covers a complete range of issues dealing with floating point numbers The Floating-Point Guide :Comparison.

We can find a somewhat more practical article in Floating-point tolerances revisited and notes there is *absolute tolerance* test, which boils down to this in C++:

```
bool absoluteToleranceCompare(double x, double y)
{
    return std::fabs(x - y) <= std::numeric_limits<double>::epsilon() ;
}
```

and *relative tolerance* test:

```
bool relativeToleranceCompare(double x, double y)
{
    double maxXY = std::max( std::fabs(x) , std::fabs(y) ) ;
    return std::fabs(x - y) <= std::numeric_limits<double>::epsilon()*maxXY ;
}
```

The article notes that the absolute test fails when `x` and `y` are large and fails in the relative case when they are small. Assuming he absolute and relative tolerance is the same a combined test would look like this:

```
bool combinedToleranceCompare(double x, double y)
{
    double maxXYOne = std::max( { 1.0, std::fabs(x) , std::fabs(y) } ) ;

    return std::fabs(x - y) <= std::numeric_limits<double>::epsilon()*maxXYOne
;
}
```

Share

Improve this answer

Follow

edited Mar 19, 2014 at 15:25

answered Feb 21, 2013 at 21:42

Shafik Yaghmour

158k ● 42 ● 461 ● 765

29

The portable way to get epsilon in C++ is

```
#include <limits>
std::numeric_limits<double>::epsilon()
```

Then the comparison function becomes

```
#include <cmath>
#include <limits>

bool AreSame(double a, double b) {
    return std::fabs(a - b) < std::numeric_limits<double>::epsilon();
}
```

Share

Improve this answer

Follow

edited Aug 8, 2010 at 0:44

answered Sep 1, 2008 at 9:59

Chris de Vries
57.6k ● 6 ● 33 ● 27

38 You'll want a multiple of that epsilon most likely. – user7116 Aug 8, 2010 at 1:41

11 Can't you just use std::abs? AFAIK, std::abs is overloaded for doubles as well. Please warn me if I'm wrong. – kolistivra Sep 25, 2010 at 8:41

3 @kolistivra, you are wrong. The 'f' in 'fabs' does not mean the type float. You're probably thinking of the C functions fabsf() and fabsl(). – jcoffland Jan 24, 2012 at 9:21

14 Actually for reasons outlined in Bruce's article **epsilon changes** as the floating point value gets bigger. See the part where he says ***"For numbers larger than 2.0 the gap between floats grows larger and if you compare floats using FLT_EPSILON then you are just doing a more-expensive and less-obvious equality check."*** – bobobobo Jan 19, 2013 at 1:07

5 i know this is old but std::abs is overloaded for floating point types in cmath. – mholzmann Jun 30, 2013 at 18:45

27

I ended up spending quite some time going through material in this great thread. I doubt everyone wants to spend so much time so I would highlight the summary of what I learned and the solution I implemented.

Quick Summary

1. Is 1e-8 approximately same as 1e-16? If you are looking at noisy sensor data then probably yes but if you are doing molecular simulation then may be not! Bottom line: You always need to think of *tolerance* value in context of specific function call and not just make it generic app-wide hard-coded constant.
2. For general library functions, it's still nice to have parameter with *default tolerance*. A typical choice is `numeric_limits::epsilon()` which is same as FLT_EPSILON in float.h. This is however problematic because epsilon for comparing values like 1.0 is not same as epsilon for values like 1E9. The FLT_EPSILON is defined for 1.0.
3. The obvious implementation to check if number is within tolerance is `fabs(a-b) <= epsilon` however this doesn't work because default epsilon is defined for 1.0. We need to scale epsilon up or down in terms of a and b.
4. There are two solution to this problem: either you set epsilon proportional to `max(a,b)` or you can get next representable numbers around a and then see if b falls into that range. The former is called "relative" method and later is called ULP method.

5. Both methods actually fails anyway when comparing with 0. In this case, application must supply correct tolerance.

Utility Functions Implementation (C++11)

```
//implements relative method - do not use for comparing with zero
//use this most of the time, tolerance needs to be meaningful in your context
template<typename TReal>
static bool isApproximatelyEqual(TReal a, TReal b, TReal tolerance =
std::numeric_limits<TReal>::epsilon())
{
    TReal diff = std::fabs(a - b);
    if (diff <= tolerance)
        return true;

    if (diff < std::fmax(std::fabs(a), std::fabs(b)) * tolerance)
        return true;

    return false;
}

//supply tolerance that is meaningful in your context
//for example, default tolerance may not work if you are comparing double with
float
template<typename TReal>
static bool isApproximatelyZero(TReal a, TReal tolerance =
std::numeric_limits<TReal>::epsilon())
{
    if (std::fabs(a) <= tolerance)
        return true;
    return false;
}


//use this when you want to be on safe side
//for example, don't start rover unless signal is above 1
template<typename TReal>
static bool isDefinitelyLessThan(TReal a, TReal b, TReal tolerance =
std::numeric_limits<TReal>::epsilon())
{
    TReal diff = a - b;
    if (diff < tolerance)
        return true;

    if (diff < std::fmax(std::fabs(a), std::fabs(b)) * tolerance)
        return true;

    return false;
}
template<typename TReal>
static bool isDefinitelyGreaterThan(TReal a, TReal b, TReal tolerance =
std::numeric_limits<TReal>::epsilon())
{
    TReal diff = a - b;
    if (diff > tolerance)
        return true;

    if (diff > std::fmax(std::fabs(a), std::fabs(b)) * tolerance)
        return true;
```

```
    return false;
}

//implements ULP method
//use this when you are only concerned about floating point precision issue
//for example, if you want to see if a is 1.0 by checking if its within
//10 closest representable floating point numbers around 1.0.
template<typename TReal>
static bool isWithinPrecisionInterval(TReal a, TReal b, unsigned int
interval_size = 1)
{
    TReal min_a = a - (a - std::nextafter(a,
std::numeric_limits<TReal>::lowest())) * interval_size;
    TReal max_a = a + (std::nextafter(a, std::numeric_limits<TReal>::max()) -
a) * interval_size;

    return min_a <= b && max_a >= b;
}
```

Share

Improve this answer

Follow

edited Dec 29, 2019 at 8:54

answered Dec 31, 2016 at 4:35

Shital Shah

68.4k • 20 • 256 • 198

1 `isDefinitelyLessThan` checks `diff < tolerance`, which means a and b are almost equal (and so a is not definitely less than b). Doesn't it make more sense to check diff > tolerance in both cases? Or perhaps add an `orEqualTo` argument which controls whether the approximate equality check should return true or not. – Matt Chambers Apr 9, 2020 at 21:30

For less than and greater than relationships, we need to use `<` and `>`. – Shital Shah Aug 2, 2020 at 5:48

1 isDefinitelyLessThan(100,100) returns true? @Shital Shah – ben Jan 9, 2023 at 2:30

15

The code you wrote is bugged :

```
return (diff < EPSILON) && (-diff > EPSILON);
```

The correct code would be :

```
return (diff < EPSILON) && (diff > -EPSILON);
```

(...and yes this is different)

I wonder if fabs wouldn't make you lose lazy evaluation in some case. I would say it depends on the compiler. You might want to try both. If they are equivalent in average, take the implementation with fabs.

If you have some info on which of the two float is more likely to be bigger than then other, you can play on the order of the comparison to take better advantage of the lazy evaluation.

Finally you might get better result by inlining this function. Not likely to improve much though...

Edit: OJ, thanks for correcting your code. I erased my comment accordingly

Share

Improve this answer

Follow

edited Aug 20, 2008 at 5:20

answered Aug 20, 2008 at 4:32

fulmicoton

15.9k ● 10 ● 55 ● 74

The question has been edited to be correct now. Both `return (diff < EPSILON) && (diff > -EPSILON);` and `return (diff < EPSILON) && (-diff < EPSILON);` are equivalent and both correct. – Gabriel Staples Nov 26, 2020 at 5:50

14

> `return fabs(a - b) < EPSILON;

This is fine if:

- the order of magnitude of your inputs don't change much
- very small numbers of opposite signs can be treated as equal

But otherwise it'll lead you into trouble. Double precision numbers have a resolution of about 16 decimal places. If the two numbers you are comparing are larger in magnitude than EPSILON*1.0E16, then you might as well be saying:

```
return a==b;
```

I'll examine a different approach that assumes you need to worry about the first issue and assume the second is fine your application. A solution would be something like:

```
#define VERYSMALL  (1.0E-150)
#define EPSILON    (1.0E-8)
bool AreSame(double a, double b)
{
    double absDiff = fabs(a - b);
    if (absDiff < VERYSMALL)
    {
        return true;
    }

    double maxAbs  = max(fabs(a) - fabs(b));
```

```
    return (absDiff/maxAbs) < EPSILON;
}
```

This is expensive computationally, but it is sometimes what is called for. This is what we have to do at my company because we deal with an engineering library and inputs can vary by a few dozen orders of magnitude.

Anyway, the point is this (and applies to practically every programming problem): Evaluate what your needs are, then come up with a solution to address your needs -- don't assume the easy answer will address your needs. If after your evaluation you find that `fabs(a-b) < EPSILON` will suffice, perfect -- use it! But be aware of its shortcomings and other possible solutions too.

Share Improve this answer Follow

answered Sep 1, 2008 at 8:00

Kevin

3 Aside from the typos (s/-/,/ missing comma in fmax()), this implementation has a bug for numbers near zero that are within EPSILON, but not quite VERYSMALL yet. E.g., AreSame(1.0E-10, 1.0E-9) reports false because the relative error is huge. You get to be the hero at your company. – brlcad Oct 21, 2010 at 0:00

1 @brlcad You didn't get the point of *floating* point. 1.0E-10 and 1.0E-9 differ by the magnitude of 10. So it is true that they are not the same. *floating* point is always about *relative* errors. If you have a system which considers 1.0E-10 and 1.0E-9 as almost equal, since both are "pretty close to zero" (which sounds reasonable to humans but is nothing mathematically), then that EPSILON needs to be adjusted as appropriate for such a system. – user2261015 Nov 12, 2015 at 11:10

10

As others have pointed out, using a fixed-exponent epsilon (such as 0.0000001) will be *useless* for values away from the epsilon value. For example, if your two values are 10000.000977 and 10000, then there are **NO** 32-bit floating-point values between these two numbers -- 10000 and 10000.000977 are as close as you can possibly get without being bit-for-bit identical. Here, an epsilon of less than 0.0009 is meaningless; you might as well use the straight equality operator.

Likewise, as the two values approach epsilon in size, the relative error grows to 100%.

Thus, trying to mix a fixed point number such as 0.00001 with floating-point values (where the exponent is arbitrary) is a pointless exercise. This will only ever work if you can be assured that the operand values lie within a narrow domain (that is, close to some specific exponent), and if you properly select an epsilon value for that specific test. If you pull a number out of the air ("Hey! 0.00001 is small, so that must be

good!"), you're doomed to numerical errors. I've spent plenty of time debugging bad numerical code where some poor schmuck tosses in random epsilon values to make yet another test case work.

If you do numerical programming of any kind and believe you need to reach for fixed-point epsilons, **READ BRUCE'S ARTICLE ON COMPARING FLOATING-POINT NUMBERS**.

Comparing Floating Point Numbers

Share Improve this answer Follow

answered Sep 12, 2011 at 22:48

Steve Hollasch

2,123 ● 1 ● 20 ● 20

7

Qt implements two functions, maybe you can learn from them:

```
static inline bool qFuzzyCompare(double p1, double p2)
{
    return (qAbs(p1 - p2) <= 0.000000000001 * qMin(qAbs(p1), qAbs(p2)));
}

static inline bool qFuzzyCompare(float p1, float p2)
{
    return (qAbs(p1 - p2) <= 0.00001f * qMin(qAbs(p1), qAbs(p2)));
}
```

And you may need the following functions, since

> Note that comparing values where either p1 or p2 is 0.0 will not work, nor does comparing values where one of the values is NaN or infinity. If one of the values is always 0.0, use qFuzzyIsNull instead. If one of the values is likely to be 0.0, one solution is to add 1.0 to both values.

```
static inline bool qFuzzyIsNull(double d)
{
    return qAbs(d) <= 0.000000000001;
}

static inline bool qFuzzyIsNull(float f)
{
    return qAbs(f) <= 0.00001f;
}
```

Share

Improve this answer

Follow

edited Apr 8, 2019 at 8:55

answered Jun 25, 2018 at 3:24

Destiny

496 ● 5 ● 8

6

You have to do this processing for floating point comparison, since float's can't be perfectly compared like integer types. Here are functions for the various comparison operators.

Floating Point Equal to (`==`)

I also prefer the subtraction technique rather than relying on `fabs()` or `abs()`, but I'd have to speed profile it on various architectures from 64-bit PC to ATMega328 microcontroller (Arduino) to really see if it makes much of a performance difference.

So, let's forget about all this absolute value stuff and just do some subtraction and comparison!

Modified from [Microsoft's example here]:

```
/// @brief      See if two floating point numbers are approximately equal.
/// @param[in]  a        number 1
/// @param[in]  b        number 2
/// @param[in]  epsilon  A small value such that if the difference between the
two numbers is
///                      smaller than this they can safely be considered to be
equal.
/// @return     true if the two numbers are approximately equal, and false
otherwise
bool is_float_eq(float a, float b, float epsilon) {
    return ((a - b) < epsilon) && ((b - a) < epsilon);
}
bool is_double_eq(double a, double b, double epsilon) {
    return ((a - b) < epsilon) && ((b - a) < epsilon);
}
```

Example usage:

```
constexpr float EPSILON = 0.0001; // 1e-4
is_float_eq(1.0001, 0.99998, EPSILON);
```

I'm not entirely sure, but it seems to me some of the criticisms of the epsilon-based approach, as described in the comments below [this highly-upvoted answer], can be resolved by using a variable epsilon, scaled according to the floating point values being compared, like this:

```
float a = 1.0001;
float b = 0.99998;
float epsilon = std::max(std::fabs(a), std::fabs(b)) * 1e-4;

is_float_eq(a, b, epsilon);
```

This way, the epsilon value scales with the floating point values and is therefore never so small of a value that it becomes insignificant.

For completeness, let's add the rest:

Greater than (`>`), and less than (`<`):

```
/// @brief      See if floating point number `a` is > `b`
/// @param[in]  a        number 1
/// @param[in]  b        number 2
/// @param[in]  epsilon  a small value such that if `a` is > `b` by this
amount, `a` is considered
```

```
///             to be definitively > `b`
/// @return     true if `a` is definitively > `b`, and false otherwise
bool is_float_gt(float a, float b, float epsilon) {
    return a > b + epsilon;
}
bool is_double_gt(double a, double b, double epsilon) {
    return a > b + epsilon;
}

/// @brief      See if floating point number `a` is < `b`
/// @param[in]  a        number 1
/// @param[in]  b        number 2
/// @param[in]  epsilon  a small value such that if `a` is < `b` by this
amount, `a` is considered
///             to be definitively < `b`
/// @return     true if `a` is definitively < `b`, and false otherwise
bool is_float_lt(float a, float b, float epsilon) {
    return a < b - epsilon;
}
bool is_double_lt(double a, double b, double epsilon) {
    return a < b - epsilon;
}
```

Greater than or equal to (`>=`), and less than or equal to (`<=`)

```
/// @brief      Returns true if `a` is definitively >= `b`, and false otherwise
bool is_float_ge(float a, float b, float epsilon) {
    return a > b - epsilon;
}
bool is_double_ge(double a, double b, double epsilon) {
    return a > b - epsilon;
}

/// @brief      Returns true if `a` is definitively <= `b`, and false otherwise
bool is_float_le(float a, float b, float epsilon) {
    return a < b + epsilon;
}
bool is_double_le(double a, double b, double epsilon) {
    return a < b + epsilon;
}
```

Additional improvements:

1. A good default value for `epsilon` in C++ is `std::numeric_limits<T>::epsilon()`, which evaluates to either `0` or `FLT_EPSILON`, `DBL_EPSILON`, or `LDBL_EPSILON`. See here: https://en.cppreference.com/w/cpp/types/numeric_limits/epsilon. You can also see the `float.h` header for `FLT_EPSILON`, `DBL_EPSILON`, and `LDBL_EPSILON`.
    1. See https://en.cppreference.com/w/cpp/header/cfloat and
    2. https://www.cplusplus.com/reference/cfloat/

2. You could template the functions instead, to handle all floating point types: `float`, `double`, and `long double`, *with type checks for these types* via a `static_assert()` inside the template.
3. Scaling the `epsilon` value is a good idea to ensure it works for really large and really small `a` and `b` values. This article recommends and explains it: http://realtimecollisiondetection.net/blog/?p=89. So, you should scale epsilon by a scaling value equal to `max(1.0, abs(a), abs(b))`, as that article explains. Otherwise, as `a` and/or `b` increase in magnitude, the epsilon would eventually become so small relative to those values that it becomes lost in the floating point error. So, we scale it to become larger in magnitude like they are. However, using `1.0` as the smallest allowed scaling factor for epsilon also ensures that for really small-magnitude `a` and `b` values, epsilon itself doesn't get scaled so small that it also becomes lost in the floating point error. So, we limit the minimum scaling factor to `1.0`. This means that for an epsilon value of `std::numeric_limits<double>::epsilon()`, for instance, the resulting `epsilon_scaled` value would be `double epsilon_scaled = std::numeric_limits<double>::epsilon() * 1.0`.
4. If you want to "encapsulate" the above functions into a class, don't. Instead, wrap them up in a namespace if you like in order to namespace them. Ex: if you put all of the stand-alone functions into a namespace called `float_comparison`, then you could access the `is_eq()` function like this, for instance: `float_comparison::is_eq(1.0, 1.5);`.
5. It might also be nice to add comparisons against zero, not just comparisons between two values.
6. So, here is a better type of solution with the above improvements in place:

```
namespace float_comparison {

/// Scale the epsilon value to become large for large-magnitude a or b,
/// but no smaller than 1.0, per the explanation above, to ensure that
/// epsilon doesn't ever fall out in floating point error as a and/or b
/// increase in magnitude.
template<typename T>
static constexpr T scale_epsilon(T a, T b, T epsilon =
    std::numeric_limits<T>::epsilon()) noexcept
{
    static_assert(std::is_floating_point_v<T>, "Floating point comparisons "
        "require type float, double, or long double.");
    T scaling_factor;
    // Special case for when a or b is infinity
    if (std::isinf(a) || std::isinf(b))
    {
        scaling_factor = 0;
    }
    else
    {
        scaling_factor = std::max({(T)1.0, std::abs(a), std::abs(b)});
    }
```

```
    T epsilon_scaled = scaling_factor * std::abs(epsilon);
    return epsilon_scaled;
}

// Compare two values

/// Equal: returns true if a is approximately == b, and false otherwise
template<typename T>
static constexpr bool is_eq(T a, T b, T epsilon =
    std::numeric_limits<T>::epsilon()) noexcept
{
    static_assert(std::is_floating_point_v<T>, "Floating point comparisons "
        "require type float, double, or long double.");
    // test `a == b` first to see if both a and b are either infinity
    // or -infinity
    return a == b || std::abs(a - b) <= scale_epsilon(a, b, epsilon);
}

/*
etc. etc.:
is_eq()
is_ne()
is_lt()
is_le()
is_gt()
is_ge()
*/

// Compare against zero

/// Equal: returns true if a is approximately == 0, and false otherwise
template<typename T>
static constexpr bool is_eq_zero(T a, T epsilon =
    std::numeric_limits<T>::epsilon()) noexcept
{
    static_assert(std::is_floating_point_v<T>, "Floating point comparisons "
        "require type float, double, or long double.");
    return is_eq(a, (T)0.0, epsilon);
}

/*
etc. etc.:
is_eq_zero()
is_ne_zero()
is_lt_zero()
is_le_zero()
is_gt_zero()
is_ge_zero()
*/

} // namespace float_comparison
```

See also:

1. The macro forms of some of the functions above in my repo here: [utilities.h](#).
    1. UPDATE 29 NOV 2020: it's a work-in-progress, and I'm going to make it a separate answer when ready, but I've produced a better, scaled-epsilon

version of all of the functions in C in this file here: utilities.c. Take a look.

2. **ADDITIONAL READING** I ~~need to do~~ now have done: Floating-point tolerances revisited, by Christer Ericson. VERY USEFUL ARTICLE! It talks about scaling epsilon in order to ensure it never falls out in floating point error, even for really large-magnitude `a` and/or `b` values!

Share

Improve this answer

Follow

edited Mar 8 at 16:27

answered Nov 26, 2020 at 2:28

Gabriel Staples
51.5k ● 30 ● 273 ● 354

5

Here's proof that using `std::numeric_limits::epsilon()` is not the answer — it fails for values greater than one:

Proof of my comment above:

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

double ItoD (__int64 x) {
    // Return double from 64-bit hexadecimal representation.
    return *(reinterpret_cast<double*>(&x));
}

void test (__int64 ai, __int64 bi) {
    double a = ItoD(ai), b = ItoD(bi);
    bool close = std::fabs(a-b) < std::numeric_limits<double>::epsilon();
    printf ("%.16f and %.16f %s close.\n", a, b, close ? "are " : "are not");
}

int main()
{
    test (0x3fe0000000000000L,
          0x3fe0000000000001L);

    test (0x3ff0000000000000L,
          0x3ff0000000000001L);
}
```

Running yields this output:

```
0.5000000000000000 and 0.5000000000000001 are  close.
1.0000000000000000 and 1.0000000000000002 are not close.
```

Note that in the second case (one and just larger than one), the two input values are as close as they can possibly be, and still compare as not close. Thus, for values greater than 1.0, you might as well just use an equality test. Fixed epsilons will not save you when comparing floating-point values.

Share Improve this answer Follow

answered Jun 23, 2018 at 2:30

Steve Hollasch
2,123 ● 1 ● 20 ● 20

1 I believe `return *(reinterpret_cast<double*>(&x));` although it usually works, is in fact undefined behaviour. – Jaap Versteegh Nov 11, 2019 at 21:15

Fair point, though this code is illustrative, so sufficient to demonstrate the issue for `numeric_limits<>::epsilon` and IEEE 754 flooring point. – Steve Hollasch Nov 12, 2019 at 22:45

Also a fair point, but it's not wise imho to post on stack overflow expecting that kind of insight. The code *will* be blindly copied making it ever harder to eradicate this very common pattern -- together with the union trick-- which should just be avoided as all UD should.
– Jaap Versteegh Nov 18, 2019 at 16:35

4

Unfortunately, even your "wasteful" code is incorrect. EPSILON is the smallest value that could be added to **1.0** and change its value. The value **1.0** is very important — larger numbers do not change when added to EPSILON. Now, you can scale this value to the numbers you are comparing to tell whether they are different or not. The correct expression for comparing two doubles is:

```
if (fabs(a - b) <= DBL_EPSILON * fmax(fabs(a), fabs(b)))
{
    // ...
}
```

This is at a minimum. In general, though, you would want to account for noise in your calculations and ignore a few of the least significant bits, so a more realistic comparison would look like:

```
if (fabs(a - b) <= 16 * DBL_EPSILON * fmax(fabs(a), fabs(b)))
{
    // ...
}
```

If comparison performance is very important to you and you know the range of your values, then you should use fixed-point numbers instead.

Share

Improve this answer

Follow

edited Jul 29, 2011 at 13:55

answered Jul 29, 2011 at 13:48

Don Reba
14k ● 3 ● 50 ● 63

2 “EPSILON is the smallest value that could be added to 1.0 and change its value”: Actually, this honor goes to the successor of 0.5*EPSILON (in the default round-to-nearest mode).

blog.frama-c.com/index.php?post/2013/05/09/FLT_EPSILON – Pascal Cuoq Aug 29, 2013 at 19:05

Why do you think that `EPSILON` in the question is `DBL_EPSILON` or `FLT_EPSILON`? The problem is in your own imagination, where you substituted `DBL_EPSILON` (which indeed would be the wrong choice) into code that did not use it. – Ben Voigt Apr 21, 2015 at 3:52

@BenVoigt, you're right, it was something on my mind at the time, and I interpreted the question in that light. – Don Reba Apr 21, 2015 at 8:56

3

General-purpose comparison of floating-point numbers is generally meaningless. How to compare really depends on a problem at hand. In many problems, numbers are sufficiently discretized to allow comparing them within a given tolerance. Unfortunately, there are just as many problems, where such trick doesn't really work. For one example, consider working with a Heaviside (step) function of a number in question (digital stock options come to mind) when your observations are very close to the barrier. Performing tolerance-based comparison wouldn't do much good, as it would effectively shift the issue from the original barrier to two new ones. Again, there is no general-purpose solution for such problems and the particular solution might require going as far as changing the numerical method in order to achieve stability.

Share Improve this answer Follow

answered Aug 28, 2008 at 11:55

Alexander L Belikoff

2

My class based on previously posted answers. Very similar to Google's code but I use a bias which pushes all NaN values above 0xFF000000. That allows a faster check for NaN.

This code is meant to demonstrate the concept, not be a general solution. Google's code already shows how to compute all the platform specific values and I didn't want to duplicate all that. I've done limited testing on this code.

```
typedef unsigned int   U32;
//  Float           Memory          Bias (unsigned)
//  -----           ------          ---------------
//   NaN            0xFFFFFFFF      0xFF800001
//   NaN            0xFF800001      0xFFFFFFFF
//  -Infinity       0xFF800000      0x00000000 ---
//  -3.40282e+038   0xFF7FFFFF      0x00000001    |
//  -1.40130e-045   0x80000001      0x7F7FFFFF    |
//  -0.0            0x80000000      0x7F800000    |--- Valid <= 0xFF000000.
//   0.0            0x00000000      0x7F800000    |    NaN > 0xFF000000
//   1.40130e-045   0x00000001      0x7F800001    |
//   3.40282e+038   0x7F7FFFFF      0xFEFFFFFF    |
//   Infinity       0x7F800000      0xFF000000 ---
//   NaN            0x7F800001      0xFF000001
```

```
//   NaN            0x7FFFFFFF      0xFF7FFFFF
//
//   Either value of NaN returns false.
//   -Infinity and +Infinity are not "close".
//   -0 and +0 are equal.
//
class CompareFloat{
public:
    union{
        float     m_f32;
        U32       m_u32;
    };
    static bool   CompareFloat::IsClose( float A, float B, U32 unitsDelta = 4 )
                  {
                      U32    a = CompareFloat::GetBiased( A );
                      U32    b = CompareFloat::GetBiased( B );

                      if ( (a > 0xFF000000) || (b > 0xFF000000) )
                      {
                          return( false );
                      }
                      return( (static_cast<U32>(abs( a - b ))) < unitsDelta );
                  }
    protected:
    static U32    CompareFloat::GetBiased( float f )
                  {
                      U32    r = ((CompareFloat*)&f)->m_u32;

                      if ( r & 0x80000000 )
                      {
                          return( ~r - 0x007FFFFF );
                      }
                      return( r + 0x7F800000 );
                  }
};
```

Share Improve this answer Follow

answered Jan 26, 2012 at 21:01

WaterbugDesign

29 ● 1

1

I'd be very wary of any of these answers that involves floating point subtraction (e.g., fabs(a-b) < epsilon). First, the floating point numbers become more sparse at greater magnitudes and at high enough magnitudes where the spacing is greater than epsilon, you might as well just be doing a == b. Second, subtracting two very close floating point numbers (as these will tend to be, given that you're looking for near equality) is exactly how you get catastrophic cancellation.

While not portable, I think grom's answer does the best job of avoiding these issues.

Share Improve this answer Follow

answered Aug 8, 2010 at 0:59

Boojum

6,692 ● 1 ● 32 ● 34

1 +1 for good information. However, I fail to see how you could mess up the equality comparison by increasing the relative error; IMHO the error becomes significant only in the result of the subtraction, however it's order of magnitude relative to that of the two operands being subtracted should still be reliable enough to judge equality. Unless of the resolution needs to be higher overall, but in that case the only solution is to move to a floating point representation with more significant bits in the mantissa. – sehe May 17, 2011 at 9:10

Subtracting two nearly-equal numbers does NOT lead to catastrophic cancellation -- in fact, it does not introduce any error at all (q.v. Sterbenz's Theorem). Catastrophic cancellation occurs earlier, during the calculation of `a` and `b` themselves. There is absolutely no problem with using floating point subtraction as part of a fuzzy comparison (though as others have said, an absolute epsilon value may or may not be appropriate for a given use case.) – Sneftel May 26, 2018 at 12:54

1

Found another interesting implementation on:
https://en.cppreference.com/w/cpp/types/numeric_limits/epsilon

```
#include <cmath>
#include <limits>
#include <iomanip>
#include <iostream>
#include <type_traits>
#include <algorithm>



template<class T>
typename std::enable_if<!std::numeric_limits<T>::is_integer, bool>::type
    almost_equal(T x, T y, int ulp)
{
    // the machine epsilon has to be scaled to the magnitude of the values used
    // and multiplied by the desired precision in ULPs (units in the last
place)
    return std::fabs(x-y) <= std::numeric_limits<T>::epsilon() * std::fabs(x+y)
* ulp
        // unless the result is subnormal
        || std::fabs(x-y) < std::numeric_limits<T>::min();
}

int main()
{
    double d1 = 0.2;
    double d2 = 1 / std::sqrt(5) / std::sqrt(5);
    std::cout << std::fixed << std::setprecision(20)
        << "d1=" << d1 << "\nd2=" << d2 << '\n';

    if(d1 == d2)
        std::cout << "d1 == d2\n";
    else
        std::cout << "d1 != d2\n";

    if(almost_equal(d1, d2, 2))
        std::cout << "d1 almost equals d2\n";
    else
```

```
        std::cout << "d1 does not almost equal d2\n";
}
```

Share Improve this answer Follow

answered Dec 19, 2019 at 2:20

Prashant Nidgunde
425 ● 4 ● 10

0

There are actually cases in numerical software where you want to check whether two floating point numbers are *exactly* equal. I posted this on a similar question

https://stackoverflow.com/a/10973098/1447411

So you can not say that "CompareDoubles1" is wrong in general.

Share

Improve this answer

Follow

edited May 23, 2017 at 12:18

Community Bot
1 ● 1

answered Jun 11, 2012 at 0:25

Michael C. Lehn
3,144 ● 2 ● 18 ● 21

> Actually a very solid reference to a good answer, though it is very specialized to limit anyone without scientific computing or numerical analysis background (I.e. LAPACK, BLAS) to not understand the completeness. Or in other words, it assumes you've read something like Numerical Recipes introduction or Numerical Analysis by Burden & Faires. – mctylr Aug 9, 2013 at 19:47

0

It depends on how precise you want the comparison to be. If you want to compare for exactly the same number, then just go with ==. (You almost never want to do this unless you actually want exactly the same number.) On any decent platform you can also do the following:

```
diff= a - b; return fabs(diff)<EPSILON;
```

as `fabs` tends to be pretty fast. By pretty fast I mean it is basically a bitwise AND, so it better be fast.

And integer tricks for comparing doubles and floats are nice but tend to make it more difficult for the various CPU pipelines to handle effectively. And it's definitely not faster on certain in-order architectures these days due to using the stack as a temporary storage area for values that are being used frequently. (Load-hit-store for those who care.)

Share

Improve this answer

edited Jul 3, 2012 at 14:49

answered Aug 20, 2008 at 16:11

Follow

user142162

Mat Noguchi
1,080 ● 6 ● 7

0

In terms of the scale of quantities:

If `epsilon` is the small fraction of the magnitude of quantity (i.e. relative value) in some certain physical sense and `A` and `B` types is comparable in the same sense, than I think, that the following is quite correct:

```
#include <limits>
#include <iomanip>
#include <iostream>

#include <cmath>
#include <cstdlib>
#include <cassert>

template< typename A, typename B >
inline
bool close_enough(A const & a, B const & b,
                  typename std::common_type< A, B >::type const & epsilon)
{
    using std::isless;
    assert(isless(0, epsilon)); // epsilon is a part of the whole quantity
    assert(isless(epsilon, 1));
    using std::abs;
    auto const delta = abs(a - b);
    auto const x = abs(a);
    auto const y = abs(b);
    // comparable generally and |a - b| < eps * (|a| + |b|) / 2
    return isless(epsilon * y, x) && isless(epsilon * x, y) && isless((delta +
delta) / (x + y), epsilon);
}

int main()
{
    std::cout << std::boolalpha << close_enough(0.9, 1.0, 0.1) << std::endl;
    std::cout << std::boolalpha << close_enough(1.0, 1.1, 0.1) << std::endl;
    std::cout << std::boolalpha << close_enough(1.1,    1.2,    0.01) <<
std::endl;
    std::cout << std::boolalpha << close_enough(1.0001, 1.0002, 0.01) <<
std::endl;
    std::cout << std::boolalpha << close_enough(1.0, 0.01, 0.1) << std::endl;
    return EXIT_SUCCESS;
}
```

Share

Improve this answer

Follow

edited Aug 29, 2013 at 18:33

answered Aug 29, 2013 at 18:06

Tomilov Anatoliy
16.6k ● 11 ● 72 ● 182

I use this code:

```
bool AlmostEqual(double v1, double v2)
    {
        return (std::fabs(v1 - v2) < std::fabs(std::min(v1, v2)) *
std::numeric_limits<double>::epsilon());
    }
```

Share Improve this answer Follow

answered May 26, 2017 at 22:11

Chunde Huang
385 ● 4 ● 9

2 That's not what `epsilon` is for. – Sneftel May 26, 2018 at 12:46

1 Why not? Can you explain it? – debuti Sep 11, 2018 at 13:32

2 @debuti `epsilon` is merely the distance between 1 and the next representable number after 1. At best, that code is just trying to check whether the two numbers are *exactly* equal to each other, but because non-powers of 2 are being multiplied by `epsilon`, it isn't even doing that correctly. – Sneftel Jun 13, 2019 at 14:03

2 Oh, and `std::fabs(std::min(v1, v2))` is incorrect -- for negative inputs it picks the one with the larger magnitude. – Sneftel Jun 13, 2019 at 15:27

In a more generic way:

```
template <typename T>
bool compareNumber(const T& a, const T& b) {
    return std::abs(a - b) < std::numeric_limits<T>::epsilon();
}
```

Note:
As pointed out by @SirGuy, this approach is flawed. I am leaving this answer here as an example not to follow.

Share

Improve this answer

Follow

edited Aug 18, 2020 at 14:48

answered Sep 16, 2016 at 3:54

André Sousa
558 ● 6 ● 10

4 This method has many weaknesses, like if the numbers `a` and `b` are already smaller than `epsilon()` there difference may still be significant. Conversely if the numbers are very large then even a couple of bits of error will make the comparison fail even if you did want the numbers to be considered equal. This answer is exactly the type of "generic" comparison algorithm that you want to avoid. – SirGuy Sep 22, 2016 at 2:05

@SirGuy How is this any different from the answer 3 posts down with 100+ upvotes? – algae Aug 18, 2020 at 11:33

@algae if you're asking why I put this comment on this answer and not the same one with 100+ upvotes then I hadn't noticed is why. – SirGuy Aug 18, 2020 at 13:41

0

I use this code. Unlike the above answers this allows one to give a `abs_relative_error` that is explained in the comments of the code.

The first version compares complex numbers, so that the error can be explained in terms of the angle between two "vectors" of the same length in the complex plane (which gives a little insight). Then from there the correct formula for two real numbers follows.

https://github.com/CarloWood/ai-utils/blob/master/almost_equal.h

The latter then is

```
template<class T>
typename std::enable_if<std::is_floating_point<T>::value, bool>::type
   almost_equal(T x, T y, T const abs_relative_error)
{
  return 2 * std::abs(x - y) <= abs_relative_error * std::abs(x + y);
}
```

where `abs_relative_error` is basically (twice) the absolute value of what comes closest to being defined in the literature: a relative error. But that is just the choice of the name.

What it really is seen most clearly in the complex plane I think. If |x| = 1, and y lays in a circle around x with diameter `abs_relative_error`, then the two are considered equal.

Share Improve this answer Follow

answered Feb 24, 2022 at 13:40

Carlo Wood

6,663 ● 2 ● 43 ● 54

0

I use the following function for floating-point numbers comparison:

```
bool approximatelyEqual(double a, double b)
{
  return fabs(a - b) <= ((fabs(a) < fabs(b) ? fabs(b) : fabs(a)) *
std::numeric_limits<double>::epsilon());
}
```

Share Improve this answer Follow

answered Sep 4, 2022 at 13:44

Athanasios Salamanis
346 ● 3 ● 6

-1

```
/// testing whether two doubles are almost equal. We consider two doubles
/// equal if the difference is within the range [0, epsilon).
///
/// epsilon: a positive number (supposed to be small)
///
/// if either x or y is 0, then we are comparing the absolute difference to
/// epsilon.
/// if both x and y are non-zero, then we are comparing the relative difference
/// to epsilon.
bool almost_equal(double x, double y, double epsilon)
{
    double diff = x - y;
    if (x != 0 && y != 0){
        diff = diff/y;
    }

    if (diff < epsilon && -1.0*diff < epsilon){
        return true;
    }
    return false;
}
```

I used this function for my small project and it works, but note the following:

Double precision error can create a surprise for you. Let's say epsilon = 1.0e-6, then 1.0 and 1.000001 should NOT be considered equal according to the above code, but on my machine the function considers them to be equal, this is because 1.000001 can not be precisely translated to a binary format, it is probably 1.0000009xxx. I test it with 1.0 and 1.0000011 and this time I get the expected result.

Share
Improve this answer
Follow

edited Mar 4, 2014 at 3:06

answered Mar 4, 2014 at 2:58

Murphy78
49 ● 1 ● 4

-1

You cannot compare two `double` with a fixed `EPSILON`. Depending on the value of `double`, `EPSILON` varies.

A better double comparison would be:

```
bool same(double a, double b)
{
  return std::nextafter(a, std::numeric_limits<double>::lowest()) <= b
    && std::nextafter(a, std::numeric_limits<double>::max()) >= b;
}
```

Share Improve this answer Follow

answered Feb 6, 2016 at 18:07

Daniel Laügt
1,147 ● 1 ● 12 ● 18

-2

My way may not be correct but useful

Convert both float to strings and then do string compare

```
bool IsFlaotEqual(float a, float b, int decimal)
{
    TCHAR form[50] = _T("");
    _stprintf(form, _T("%%.%df"), decimal);


    TCHAR a1[30] = _T(""), a2[30] = _T("");
    _stprintf(a1, form, a);
    _stprintf(a2, form, b);

    if( _tcscmp(a1, a2) == 0 )
        return true;

    return false;

}
```

operator overlaoding can also be done

Share Improve this answer Follow

answered Dec 3, 2013 at 8:03

Vijay
2,049 ● 4 ● 28 ● 35

+1: hey, I'm not going to do game programming with this, but the idea of round-tripping floats came up several times in Bruce Dawson's blog (treatise? :D) on the issue, and if you're trapped in a room and someone puts a gun to your head and says "hey you have to compare two floats to within X significant figures, you have 5 minutes, GO!" this is probably one to consider. ;) – shelleybutterfly May 25, 2014 at 2:04

@shelleybutterfly Then again the question was for the most *efficient* way of comparing two floating point numbers. – Tommy Andersen Aug 13, 2014 at 11:11

@TommyA lol perhaps, but I bet round-tripping was downvoted for reasons not efficiency-related. Though my intuition is that it would be rather inefficient compared to HW fp math but also says that the algoritihms in software fp are unlikely to have a bigO difference at least. I eagerly await the analysis you did showing the efficiency concerns in that case are significant. Besides, sometimes less-than-optimal can still be a valuable answer, and as it was downvoted–despite being a valid technique that was even mentioned by Dawson's blog on the subject, so I thought it deserved an upvote. – shelleybutterfly Aug 24, 2014 at 8:04

-3

This is another solution with lambda:

```
#include <cmath>
#include <limits>

auto Compare = [](float a, float b, float epsilon =
std::numeric_limits<float>::epsilon()){ return (std::fabs(a - b) <= epsilon);
};
```

Share Improve this answer Follow

answered Jan 26, 2020 at 15:22

Amir Saniyan

13.7k ● 22 ● 101 ● 141

This is exactly the same as many of the other answers except that it's a lambda and has no explanation, so this doesn't add much value as an answer. – stijn Apr 29, 2020 at 12:10

1 2 Next

Highly active question. Earn 10 reputation (not counting the association bonus) in order to answer this question. The reputation requirement helps protect this question from spam and non-answer activity.