TAMING THE FLOATING POINT BEAST

CHRIS LOMONT

ABSTRACT. Comparing floating point numbers for near equality has been a well discussed topic. This note covers the relevant points, and then introduces a few new, robust, high performance ways to do these comparisons. Applications range from realtime raytracing to video games to CAD systems. In particular branchless comparisons are designed which are faster than previous methods with no loss of numerical quality.

1. INTRODUCTION

Imagine the surprise when a new programmer tries to run C++ code similar to Listing 1 and gets the unexpected result[1] that `0.2*5` is not equal to `1.0`. The first

```
#include <iostream>
int main(void)
    {
    float a = 0.2f;
    a = 5*a;
    if (1.0f == a)
        std::cout << "Equal\n";
    else
        std::cout << "Not equal\n";
    return 0;
    }
```
<center>Listing 1.</center>

thought in the new programmer's mind is that something was mistyped, but after that is checked, the second thought is that the computer is wrong. Trying to figure out what is going on the programmer tries to print out the values with

```
std::cout << a << std::endl;
```
only to find the output is still `1`. Finally, if the programmer is clever, he[2] tries

```
std::cout << a - 1.0f << std::endl;
```
obtaining something like `1.49012e-008`, which gives a hint that perhaps `5*a` really is not `1` in the crazy world of the computer!

So, the programmer thinks of a common solution: compare the absolute value of the difference with some tolerance by replacing the `if` line with Listing 2 and all is well. Or is it? Suppose this is all the farther the programmer looks, and

---

[1]On what line did the programmer make the first mistake? The line a=0.2f. At that point **a** is not truly 0.2, but merely an approximation.

[2]I dislike writing "he or she," everywhere so I use the original English neutral "he" which *means* "he or she."

```
if (fabs(a-1.0f) < 0.000001)
```
                                Listing 2.

code like this soon litters the application, making its way into libraries created by the programmer which are then used by others. This leads to faults similar to the spectacular Ariane 5 Rocket failure, which blew up 37 seconds into flight in 1996 due to a floating point library error, costing a half billion dollars, as well as other very costly numerical errors [1].

Next the programmer tries to compute his yearly salary by multiplying his weekly paycheck amount of `a=$1500.15` by `52`. The correct answer is $78007.80, but the tolerance `0.000001` above is far too small, and in order to compare the computed result with the answer he knows is right, the required tolerance becomes `fabs(a*52-78007.80f) < 0.01`. No smaller value of the form `0.00...01` works. However `0.01` is too big for other problems. In useful library code, what value should be used?

The lesson is that the tolerance required depends on the size of the values being tested.

At this point the programmer should return to the bible: Knuth's series "The Art of Computer Programming." Volume II, Seminumerical Algorithms [5], has the following advice: a good floating point routine should claim two floats are nearly equal not if their *absolute* error is bounded by a tolerance, but if if their *relative* error is bounded by some tolerance. So we follow Knuth's advice from section 4.2.2, and code some function like Listing 3.


```
bool KnuthCompare(float aFloat, float bFloat, float relError)
    {
    // we claim they are close if either is close enough
    // Note we need <= to handle a=b=0 case, < is not good enough
    return (fabs(aFloat - bFloat) <=
            relError * max(fabs(aFloat),fabs(bFloat))
    }
```
                                Listing 3.

Note we need to be careful to catch when `a` and `b` are both exactly zero, otherwise our routine would say $0 \neq 0$. Note we changed the routine a little from Knuth to make the routine symmetric, that is,

    `KnuthCompare(a,b) == KnuthCompare(b,a)`

which fails for one of the routines in his book.

Now we have a numerically better routine. The above examples can be replaced with `KnuthCompare(52*a,78007.80,0.01)` and `KnuthCompare(a,1.0f,0.01)`. All is well.

Two problems remain: 1) `KnuthCompare` is much slower (as shown in section 4.3, up to 20 times slower!), and 2) how does one figure out a good value for `relError`?

Well, the second answer really depends on the use of the function, and a good library should allow the user to specify the tolerance. We shall work on the speeding up the algorithm, and as a side effect, we will actually get a much better way to measure "closeness" of floating point numbers!

How can this be improved? A diligent or curious programmer at this point would dig around on the internet to find out what is going on. He would find details on how floating point numbers are stored, which is crucial to this story.

2. Floating point format

2.1. **A Brief History.** Floating point numbers are stored on most modern computers in IEEE 754 format. This includes Intel x86 and IA64, AMD, Motorola 68K, IBM Power and PowerPC. Apple uses a software solution (SANE). Many RISC systems use a mix of software and hardware, like DEC Alpha, HP PA-RISC, SGI MIPS, Sun SPARC. See [2] for details.

Before the IEEE 754 spec (1985) computers used a variety of formats and many had very poor floating point support, so porting numerical code was error-prone, buggy, and generally best left to magicians. Now that most common machines have good floating point support, mortals can master the details. For a wonderful story on how this standard came to be, read Kahan's[3] article [4].

2.2. **IEEE 754 Floating Point Representation.** Floating point numbers are stored on the PC (and almost everywhere!) as 32 bit numbers;

s	E	M
bit 31	30 ← *bits* → 23	22 ← *bits* → 0

where s is a 1 bit sign (1 denotes negative, 0 positive), E is an 8 bit exponent, and M is a 23 bit mantissa. The exponent E is *biased* by 127 to accommodate positive and negative exponents. The 32 bit float format on x86 has a **normalized** mantissa, which means the most significant bit of the true mantissa is 1. This is a **packed** format, which means the leading 1 is not explicitly stored in the mantissa M, but inferred. View M as a binary number with the decimal point to the left, thus M is a value in the half-open interval $[0, 1)$. The represented floating point value x is then

$$x = (-1)^s (1.M) 2^{E-127}$$

where if M is the bitstring $m_1 m_2 m_3 \ldots m_{23}$ then $(1.M)$ is the value $1 + m_1/2 + m_2/4 + m_3/8 + \cdots + m_{23}/2^{23}$. This holds true for $1 \le E \le 254$. When $E = 0$ or $E = 255$ there may other values, explained later.

Internally the x86 uses 80 bit floating point registers, so the value of this 32-bit number is unpacked (inserting the missing 1 bit) before being used in computations. Floats are unpacked when loading into registers, and repacked when stored back to memory. The `long double`, shown below, is an 80-bit format that fills the entire register, so there is no room to unpack it. Thus the normalized 1 value is explicitly stored in the `long double` type, which is an unpacked type.

While we are at it, `double` and various `long double`s follow similar bit layouts, with the parameters shown in Table 1.

2.2.1. *Exceptions.* That is almost the whole story, except the special cases which we detail here.

(1) **Normalized numbers**: when $1 \le E \le 254$, the bit pattern represents a real number as explained above. Note there is no way to denote zero with this way, due to the implied 1 in the mantissa, so we have.....

---

[3]Kahan is called "The Father of Floating Point," since he was instrumental in creating the IEEE 754 spec.

type	float	double	Intel long double	SPARC long double
bits	32	64	80	128
sign bits	1	1	1	1
exponent bits	8	11	15	15
exponent bias	127	1023	16383	16383
mantissa bits	23	52	64	112
packed	yes	yes	no	yes
digits precision	7	16	19	34

TABLE 1. Values describing floating point formats

(2) **(Signed) zeroes**: if $E = 0$ and $M = 0$ then the value is zero. The sign bit gives ± 0, so we get two bit patterns, each representing zero. IEEE 754 requires these two bit representations to compare as equal, however.

(3) **Denormalized numbers** (also called *subnormal* numbers): If $E = 0$ and $M \neq 0$, the pattern represents a *denormalized* number. These no longer have the implied 1 in the mantissa, and represent small values. This allows the bit patterns to represent more (very small) numbers but with a loss of precision.

(4) **Infinities**: if $E = 255$ and $M = 0$, the sign bit then implies the values $\pm\infty$.

(5) **Not-a-Number** (NaNs): $E = 255$ and $M = 0.0xx$ where $xx \neq 0$ denotes a Signalling NaN, or SNaN. The mantissa $M = 0.1xx$ is termed a Quiet NaN, or QNaN. Intel only implements one, and calls it simply a QNaN.

These rules also generalize to `double` and `long double`.

These bit patterns are part of the IEEE 754 spec so that error conditions such as arithmetic overflow, underflow, taking the square root of a negative number, dividing by 0, etc., yield a result representable within the format, allowing the programmer to catch results and act appropriately.

2.3. **IEEE 754 Gotchas.** These various special numbers allow nicer handling of many math operations. For example, $\infty + 1 = \infty$, $\infty + \infty = \infty$, $\infty - \infty = \text{NaN}$, $\sqrt{-1} = \text{NaN}$. There are also weird effects: `NaN` compared to any value returns false, whether it is using `<`, `==`, or `>`. Thus for the floating variable `a` it is not true that `a==a` when `a` is `NaN`. More details are at:

oregonstate.edu/~peterseb/mth351/docs/351s2001_fp80x87.html
www-106.ibm.com/developerworks/java/library/j-jtp0114

In the case of 32-bit floating point, we can now draw the useful and easy to visualize (vertical) number line showing how the binary patterns representing IEEE 754 numbers match up to the usual number line, as shown in Table 2.

At this point it would be instructive to verify the floating point value `123.45f` has IEEE 754 binary representation `0x42F6E666` and that binary `0xABCDEF00` represents `-1.46325e-012`.

Armed with precise IEEE 754 layout knowledge, our clever programmer should try to find a better solution. And this is what is going to be done in the next section.

Value	Hex	Decimal	Name	Value	Hex	Decimal	Name
-1.#QNAN	FFFFFFFF	-1		1.#QNAN	7FFFFFFF	2147483647	
-1.#QNAN	FFFFFFFE	-2		1.#QNAN	7FFFFFFE	2147483646	
-1.#QNAN	FFFFFFFD	-3	Negative	1.#QNAN	7FFFFFFD	2147483645	Positive
...	...	...	NaNs	...	...	...	NaNs
-1.#QNAN	FF800003	-8388605		1.#QNAN	7F800003	2139095043	
-1.#QNAN	FF800002	-8388606		1.#QNAN	7F800002	2139095042	
-1.#QNAN	FF800001	-8388607		1.#QNAN	7F800001	2139095041	
-1.#INF	FF800000	-8388608	$-\infty$	1.#INF	7F800000	2139095040	$+\infty$
-3.40282e+038	FF7FFFFF	-8388609		3.40282e+038	7F7FFFFF	2139095039	
-3.40282e+038	FF7FFFFE	-8388610		3.40282e+038	7F7FFFFE	2139095038	
-3.40282e+038	FF7FFFFD	-8388611	Negative	3.40282e+038	7F7FFFFD	2139095037	Positive
...	...	...	Normalized	...	...	...	Normalized
-1.17549e-038	80800002	-2139095038		1.17549e-038	00800002	8388610	
-1.17549e-038	80800001	-2139095039		1.17549e-038	00800001	8388609	
-1.17549e-038	80800000	-2139095040		1.17549e-038	00800000	8388608	
-1.17549e-038	807FFFFF	-2139095041		1.17549e-038	007FFFFF	8388607	
-1.17549e-038	807FFFFE	-2139095042		1.17549e-038	007FFFFE	8388606	
-1.17549e-038	807FFFFD	-2139095043	Negative	1.17549e-038	007FFFFD	8388605	Positive
...	...	...	Denormalized	...	...	...	Denormalized
-4.2039e-045	80000003	-2147483645		4.2039e-045	00000003	3	
-2.8026e-045	80000002	-2147483646		2.8026e-045	00000002	2	
-1.4013e-045	80000001	-2147483647		1.4013e-045	00000001	1	
0	80000000	-2147483648	-0	0	00000000	0	+0

Table 2. The real number line, in 32-bit IEEE 754 floats. The left half is negative values; the right half is positive. Intel has another value for indeterminate, `-1.#IND FFC00000 -4194304`.

3. Initial solution

Looking at the number line in Table 2, we can see an elegant way to compare (IEEE 754) floating point numbers. We compare them as binary integers! Usually floating comparison arises from computing some value in two different ways, and these values should be equal, or nearly so. However, due to floating point roundoff and accumulated error, the two answers differ by a small amount, which corresponds to a small difference in the resulting binary bit patterns representing the answers. In other words, if both answers are positive, from looking at the number line in Table 2, we see they should be close as binary integers. This leads to the first attempt[4] in Listing 4.

```
// initial idea - incorrect though!!!
bool CompareFloatBad(float af, float bf, int maxDiff)
    { // this fails for some cases!!!
    int ai = *reinterpret_cast<int*>(&af);
    int bi = *reinterpret_cast<int*>(&bf);
    int diff = ai - bi;
    if (abs(diff) < maxDiff)
        return true;
    return false;
    } // CompareFloatBad
```
Listing 4.

This is a good place to mention two caveats for the routines we will develop:
- We require `sizeof(float) == sizeof(int)` in order to access the `float`s as `int`s.

---

[4]While working on this, I found the same initial idea from Bruce Dawson[3].

- These routines do not have the correct behavior for `NaN` and infinities, so as long as your floating point values represent finite, real values, these faster routines will work.

Listing 4 does not take into account negative values, and will fail to claim the positive zero and the negative zero are close. We want to make small negative and positive numbers appear close as `int`s, and to make the two representations of ± 0 be equal. Looking again at the number line, notice that the negative values, treated as integers, go "the other way". So to reverse their order, we can negate them with

```
if (ai < 0)
    ai = -ai;
```

but this still leaves a big gap between positive 0 and negative 0. To make these two numbers equal as integers, and which also makes small negative floats close to small positive floats, we finally notice that the code

```
if (ai < 0)
    ai = 0x80000000 - ai;
```

completely makes adjacent floating values be adjacent as integers. This leads to listing 5, which I name after Bruce Dawson[3], since he thought of it before me.

```
bool DawsonCompare(float af, float bf, int maxDiff)
    {
    int ai = *reinterpret_cast<int*>(&af);
    int bi = *reinterpret_cast<int*>(&bf);
    if (ai < 0)
        ai = 0x80000000 - ai;
    if (bi < 0)
        bi = 0x80000000 - bi;
    int diff = ai - bi;
    if (abs(diff) < maxDiff)
        return true;
    return false;
    }
```

Listing 5.

So now if you want two `float` values to be within `100` possible values of each other, you can check that nicely. And this method scales automatically over the whole range of floating values.

Timing this code, which is placed in the `(MID1,FINAL1)` block of table 3, we find that Dawson's comparison is faster than the Knuth code in Listing 3, and is often faster than the original simple `fabs(...) < tolerance` version! So integer comparison is on the right track!

4. Optimizing the solution

Dawson's comparison in is a fairly robust and fast floating point comparison routine. Question: can we make it faster yet? The main thing we attempt is removing the branching, that is, remove the necessity for the `if` statements, as well as some "hidden" branches. Branches can slow down code dramatically on modern processors. To see the results, look over section 4.3 and Table 3.

4.1. **The final block.** To remove the branching, start with the final three lines in Listing 5, which we label `FINAL1`, since it is the first "final" block we modify. We

```
if (abs(diff) <= maxDiff)
    return true;
return false;
```
<div align="center">Listing FINAL1</div>

want to remove the `if` statement. Note there is a hidden `if` in the `abs` statement, which is usually a template or macro that expands to an `if` statement. Our first try is to expand the `abs(diff)<=maxDiff` to the mathematically equivalent code in Listing `FINAL2`. Is this faster? Sadly, the timings in Table 3 show it is not. What

```
return (-maxDiff <= diff) && (diff <= maxDiff);
```
<div align="center">Listing FINAL2</div>

happened? Well, Listing `FINAL2` now has two comparisons to do, even through it removed the obvious and hidden `if`. However, due to C++ requiring short-circuited logical `AND`, [8] the `&&` above hides another test and branch at the assembly level on most machines, which we also remove.

Note that `-maxDiff <= diff` is equivalent to `0<=maxDiff+diff`, which is equivalent to the high bit being clear in `int v1 = maxDiff + diff`. Converting both inequalities like this and bitwise `OR`ing gives a value which should have the high bit clear, which means non-negative. This is carried out in Listing `FINAL3`. At

```
int v1 = maxDiff + diff;
int v2 = maxDiff - diff;
return (v1|v2) >= 0;
```
<div align="center">Listing FINAL3</div>

the assembly level this requires an add, a subtract, an `OR`, and a compare. No branches here! Thus we fulfilled our requirement. (Some architectures may still create a branch, but on x86 machines no branches exist under popular compilers). As expected this time, this is faster than `FINAL2`, and beats `FINAL1` in some cases, giving the fastest overall routine when we optimize the middle block.

4.2. **The Middle Block.** Now we take the original computation of `diff` and try to do the same tricks. The original block in Listing `MID1` has two branches, inde-

```
if (ai < 0)
    ai = 0x80000000 - ai;
if (bi < 0)
    bi = 0x80000000 - bi;
int diff = ai - bi;
```
<div align="center">Listing MID1</div>

pendently called if `ai` or `bi` is negative, which might be quite likely in a library. Recall we needed branches because if `ai` or `bi` were negative, their ordering was incorrect when we computed `diff`.

However, since we only need the absolute value of `diff`, we really only need to branch if `a` and `b` have *opposite* signs. If they have the same sign we can ignore reversing them. Also, it does not matter mathematically which one we reverse, since we only need their difference and can ignore if it is positive or negative. (Think about this until you see why this is true. In short, the code only needs to compute ±`diff`). Thus we proceed as follows: compute the difference, and if `a` and `b` had opposite signs, only *then* compute a new difference where we reverse `a` and leave `b` alone. This leads to the second middle block in Listing `MID2`.

```
int diff = ai - bi; /* assumes both same sign */
if (0x80000000 & (ai^bi)) // sign differed, so...
    diff = (0x80000000 - ai) - bi; // reverse one
```
<div align="center">Listing MID2</div>

Aha! We got one right. `MID2` is faster than `MID1` across the board (Table 3). Now we are down to one `if`.

To remove this last `if`, first note we probably need to compute both `ai-bi` and `0x80000000-ai-bi`. We want to create some sort of mask or shift or multiply from the condition on the `if`, which was (`0x80000000&(ai^bi)`). This expression is `0` if `a` and `b` have the same sign, else is `0x80000000`. One idea is to shift this bit down to give a value of `0` or `32`, and use this to shift the 32-bit values for either `ai-bi` or `0x80000000-ai-bi` to either give `0` or the desired value in `diff`. However, a 32 bit shift fails on most x86 architectures (the operand produced is ignored by the CPU without warning - try it!), so unfortunately we are forced to use a 16-bit shift and apply it twice. The resulting code is Listing `MID3`. We had to treat `ai` as an

```
// create 16 bit shift if signs differ
int test = (0x80000000 & (ai^bi))>>(31-4);
assert((0 == test) || (16 == test));
int test2 = 16-test;
int diff = (( ((unsigned int)(ai))>>test)>>test) +
    (( (unsigned int)(0x80000000 - ai) >>test2)>>test2) -
    bi;
```
<div align="center">Listing MID3</div>

`unsigned int` to cause the shift to operate properly, since *the C++ standard does not define right shifting for* `signed int`*s*! We used the obsolete version of a cast to make the print nicely - it is preferable to use `reinterpret_cast<unsigned int>`.

`MID3` has the advantage that performance can be constant[5] across all inputs, as shown in Table 3. It is slower than `MID2` in some cases and faster in others. Thus we trudge on....

A slight variation, Listing `MID4`, creates `test2` using an `XOR` instead of a subtract, which might be faster on some machines. On those tested it has almost exactly the same performance.

So we try another tack to choosing which `diff` value to use. This idea is to create masks of `0x00000000` and `0xFFFFFFFF` to `AND` against the two possible values. The first way we create these masks is to shift the bit denoting `ai` and `bi` have different

---

[5]Up to profiler error and choice of FINAL block.

```
int test = (0x80000000 & (ai^bi))>>(31-4);
int test2 = 16^test;
assert((0 == test) || (16 == test));
int diff = (( ((unsigned int)(ai))>>test)>>test) +
    (( (unsigned int)(0x80000000 - ai) >>test2)>>test2) -
    bi;
```

<div align="center">Listing MID4</div>

signs down to the lowest position, resulting in 0 or 1. Subtracting 1 then gives the masks, with resulting code in Listing MID5. MID5 has the fastest overall worst

```
int test = (((unsigned int)(ai^bi))>>31)-1;
assert((0 == test) || (0xFFFFFFFF == test));
int diff = (((0x80000000 - ai)&(~test)) | (ai & test)) - bi;
```

<div align="center">Listing MID5</div>

case performance yet, although for some cases MID2 is faster. However, MID5 allows constant time across all inputs and is branch free, which is useful.

The final variation, MID6, replaces the mask creation with one requiring fewer basic operations by noting that shift of the signed value creates the mask automatically **on some architectures and compilers**. This exploits the unspecified behavior in C++ when shifting signed ints. This is the highest performing worst

```
int test = (ai^bi)>>31;
assert((0 == test) || (0xFFFFFFFF == test));
int diff = (((0x80000000 - ai)&(test)) | (ai & (~test))) - bi;
```

<div align="center">Listing MID6</div>

case code in this paper, but the code is less portable, since the C++ standard does not define what will happen when shifting a negative int. On Microsoft's Visual Studio environment (and perhaps all x86 compilers?) the code works since these compilers guarantee the type of shifting required.

4.3. **The results.** Table 3 gives the result of the many comparison versions on various x86 platforms. Although only two computers have results here (one AMD and one Intel), many other AMD and Intel PCs were tested with very similar performance results. These two were selected to indicate the relative performance of the different comparison methods.

The left column shows the performance of the naive fabs solution (Listing 2), the KnuthCompare (Listing 3), and a Dummy test that is merely a function that always returns true to help factor out testing framework overhead. The other 18 versions on the right grid are those resulting from the various code snippets labelled along the grid edges.

The testing used 5 arrays of 10,000 float comparisons to be done; each array was processed 1,000 times. From these 1,000 runs, the best performance was taken (which in each case was within 1 cycle from the average over the entire 1,000). The 5 different arrays had 0, 25, 50, 75, and 100 percent of the comparisons using different signs, which led to varying performance in some routines since some routines use branching based on sign changes. In each cell, there is a high and low number,

which gives the best and worst performance over the 5 types of arrays, and shows how the routine performs from 0 percent sign mixing through 100 percent sign mixing. Values are in clock cycles.

PC 1			MID1	MID2	MID3	MID4	MID5	MID6
Dummy	16 16	FINAL1	34 41	25 34	28 28	30 30	30 30	25 26
Knuth	35 508	FINAL2	47 57	31 41	35 39	36 39	35 38	31 35
fabs	41 92	FINAL3	35 42	**25** **35**	29 29	28 29	**28** **29**	**26** **26**
PC 2			MID1	MID2	MID3	MID4	MID5	MID6
Dummy	96 96	FINAL1	116 130	104 115	112 112	112 112	108 108	104 104
Knuth	136 6575	FINAL2	121 136	107 119	123 126	120 125	111 113	108 111
fabs	116 1769	FINAL3	115 126	**104** **112**	108 108	108 108	**108** **108**	**104** **104**
PC 3			MID1	MID2	MID3	MID4	MID5	MID6
Dummy	47 47	FINAL1	69 70	57 63	65 65	66 67	65 67	63 64
Knuth	73 1073	FINAL2	72 76	63 69	68 71	69 72	70 72	65 68
fabs	62 292	FINAL3	69 70	**58** **64**	66 66	67 67	**62** **62**	**61** **62**

TABLE 3. Cycle counts. PC 1 is a WinXP AMD Opteron 246 2.0 GHZ 4GB RAM, PC2 is a WinXP P4 2.66 GHZ 1 GB RAM, PC 3 is a Win98 PIII 500 MHZ 128MB RAM. Bold letters denote routines recommended as best overall across platforms.

For example, on PC 1, comparing the Knuth version to the fabs version, subtracting out the best case Dummy time, shows the best case Knuth version uses $\frac{73-47}{62-47} = 1.733$ as much time, hence is almost twice as slow. Measuring the worst case Knuth version is an order of magnitude slower. Similarly, on the Athlon, the (MID6,FINAL3) version uses at most $\frac{26-16}{41-16} = 0.4$ of the time for the naive fabs version, over twice as fast!

5. CONCLUSION

We found three routines that are useful. Listing 6 has the fastest possible best case performance across the platforms we tested, but can slow down depending on the mix of values. This is the (MID2,FINAL3) combination.

Listing 7 has the fastest constant time performance we tested, but relies on a loophole in the C++ specification. It works on Microsoft compilers. This is the (MID6,FINAL3) combination.

The final routine, Listing 8, is a fast routine that will work on a larger class of compilers, since it does not rely on the loophole. It is slightly slower than Listing 7, and has performance in between the extremes of Listing 6, and corresponds to (MID5,FINAL3). This is probably the best version overall, and the one I recommend for general use.

```
bool LomontCompare1(float af, float bf, int maxDiff)
    { // Fast, non constant time routine, portable
    int ai = *reinterpret_cast<int*>(&af);
    int bi = *reinterpret_cast<int*>(&bf);
    int diff = ai - bi; /* assumes both same sign */
    if (0x80000000 & (ai^bi)) // sign differed, so...
        diff = (0x80000000 - ai) - bi; // reverse one
    int v1 = maxDiff + diff;
    int v2 = maxDiff - diff;
    return (v1|v2) >= 0;
    }
```

Listing 6.

```
bool LomontCompare2(float af, float bf, int maxDiff)
    { // fast routine, not portable due to shifting signed int
      // works on Microsoft compilers.
    int ai = *reinterpret_cast<int*>(&af);
    int bi = *reinterpret_cast<int*>(&bf);
    int test = (ai^bi)>>31;
    assert((0 == test) || (0xFFFFFFFF == test));
    int diff = (((0x80000000 - ai)&(test)) | (ai & (~test))) - bi;
    int v1 = maxDiff + diff;
    int v2 = maxDiff - diff;
    return (v1|v2) >= 0;
    }
```

Listing 7.

```
bool LomontCompare3(float af, float bf, int maxDiff)
    { // solid, fast routine across all platforms
      // with constant time behavior
    int ai = *reinterpret_cast<int*>(&af);
    int bi = *reinterpret_cast<int*>(&bf);
    int test = (((unsigned int)(ai^bi))>>31)-1;
    assert((0 == test) || (0xFFFFFFFF == test));
    int diff = (((0x80000000 - ai)&(~test)) | (ai & test)) - bi;
    int v1 = maxDiff + diff;
    int v2 = maxDiff - diff;
    return (v1|v2) >= 0;
    }
```

Listing 8.

Finally, a few homework problems to ensure you have absorbed the material presented.

1. If your compiler supports 64-bit long integers and 64-bit doubles, create a similar routine comparing doubles. Hint: this is trivial.

2. Create similar routines for comparing floats using greater or less than comparisons with a little padding, i.e., a routine with the signature

```
bool FloatLess(float a, float b, int padding);
```

Code can be found at www.lomont.org under the Papers section. To see more fast floating point routines, see the fast inverse square root paper [7] and the fast non-linear gradient fill paper [6].

References

1. Douglas N. Arnold, *Some disasters attributable to bad numerical computing*, www.ima.umn.edu/∼arnold/disasters/disasters.html.
2. Nelson H. F. Beebe, *IEEE 754 floating-point test software*, http://www.math.utah.edu/∼beebe/software/ieee/.
3. Bruce Dawson, *Comparing floating point numbers*.
4. William Kahan, *An Interview with the Old Man of Floating-Point*, 1998, www.cs.berkeley.edu/ wkahan/ieee754status/754story.html.
5. Donald Knuth, *The Art of Computer Programming, Volume 2: Seminumerical Algorithms (3rd Edition)*, Addison-Wesley, 1997, ISBN: 0201896842.
6. Chris Lomont, *Deriving fast nonlinear gradient fills*, 2003, www.lomont.org/Math/Papers/2002/FinalPaper.pdf.
7. _____, *Fast Inverse Square Root*, 2003, www.lomont.org/Math/Papers/2003/InvSqrt.pdf.
8. Bjarne Stroustrup, *The C++ Programming Language, Special Third Edition*, Addison-Wesley Professional Computing Series, Boston, New York, 2000, ISBN: 0201700735.

Email address: chris@lomont.org
Web address: www.lomont.org
First written: Apr 2005