Learning from Math Library Testing for C

Marcel Beemster – Solid Sands

Introduction

In the process of improving SuperTest, I recently dived into its math library testing. Turns out there were some interesting observations to make. I have summarized them here in four lessons. The take-away is that our intuition about real numbers does not translate directly to floating point computations. When working with floating point, it is essential to have a model of its accuracy in mind. This white paper is my attempt to capture some intuition about floating point computations.

The fun thing about maintaining and improving a generic test suite such as SuperTest is that it encompasses so many special areas to know about. Math library testing is definitely one of them.

In my endeavours, I learned four new things:

- all binary floating point numbers have an accurate and finite decimal representation; but not the other way around
- there is a fun way to double the accurate number of digits of **Pi** using a trivial equation
- current implementations of the sine, cosine and tangent functions are crazy accurate
- that crazy accuracy serves no useful purpose

Some basic background: the C math library uses fixed size floating point numbers for its computing. Floating point ("FP") numbers have useful properties: they can represent very large and very small numbers, and their relative accuracy over the FP-domain is about the same everywhere. They also have a very bad property: a floating point number is almost always an approximation of the real value that you want to represent because FP numbers are just dots along the real number line. The values between the dots cannot be represented accurately by an FP number. Close to zero, there are very many dots. For very big values, the dots are getting sparse, with large gaps between them.

The size of the gap between two dots is determined by the number of bits used in the fractional part of the FP representation. The other part, the exponent, determines where the fractional point is placed. More fractional bits means smaller gaps. The fractional part is commonly called the mantissa.

When you have a value that sits between two such dots on the number line, a good FP implementation chooses a nearby dot to represent that value. By choosing the nearest dot, you are never more than half a gap wrong. The technical term for that is that the best accuracy you can ever hope to achieve on average is **0.5** ULP, where ULP stands for "Units in the Last Place" (where, in the usual base-**2** representation, you can also read *bit* instead of *unit*).

The First Lesson: Converting Between Binary and Decimal

Another nasty property of FP: in a binary (base-**2**) representation of FP numbers, used by computers, the dots on the FP number line are in a different position than the dots in a decimal (base-**10**) representation that is used by humans. Some dots coincide: for small whole numbers and fractions of **2**, but many do not. That means you have to be careful with conversions: you may drop some “ULP".

But we can be more accurate than that:

- Every finite binary (base-**2**) FP number can be represented by a finite base-**10** number. This is because **2** shares a prime factor with **10**.
- But not every finite base-**10** number can be represented by a finite binary number. That is because **5** (the other prime factor of **10**) is relative prime to **2**.

Here is one way to understand this. The following table lists the conversion from binary fractional numbers (left) to decimal fractions (right):

Binary	**Decimal**
0.1	0.5
0.01	0.25
0.001	0.125
0.0001	0.0625

The table goes on: for every such **1**-bit fraction in binary notation, there is a finite decimal fraction. Since every multi-bit fraction in binary notation is the addition of **1**-bit fractions, their sum in decimal notation stays finite.

But it does not work in the other direction:

Decimal	**Binary**
0.1	0.000110011001100....

So the binary notation for **1/10**th is not a finite fractional number. Its last four bits (**1100**) repeat endlessly.

Corollary 1: Be careful when converting decimal numbers to (finite) binary FP numbers.

Corollary 2: For an accurate decimal representation of a binary FP number, you may need about as many digits as you have bits in the mantissa.

The Second Lesson: Doubling the Digits of π

The second thing I learned came from the Random ASCII blog:

https://randomascii.wordpress.com/2014/10/09/intel-underestimates-error-bounds-by-1-3-quintillion/

It refers to the following handy equation:

Pi = Pi + sin(Pi)

That appears trivial because in proper maths, **sin(Pi)** equals zero. But in the computer, **Pi** is one of those values that falls in the gaps between the dots on the number line. In the machine, it is rounded to a nearby dot. Let's call the *nearest-dot-approximation* of **Pi** **M_Pi** (Machine-Pi). So if you compute **sin(M_Pi)**, you should NOT get zero as a result because **M_Pi** is not equal to **Pi**.

Here are a few lines of C-code to type in, compile and execute:

```
#include <stdio.h>
#include <math.h>
#define M_Pi    3.14159265358979323846264338327795
int main (void) {
    printf ("M_Pi      : %.31f\n", M_Pi);
    printf ("sin(M_Pi): %.31f\n", sin (M_Pi));
    printf ("Real Pi   : 3.14159265358979323846264338327795...\n");
    return 0;
}
```

The first two lines print **M_Pi** and **sin(M_Pi)**, the two right side components of the equation. Notice that the printed **M_Pi** is different from its definition: the definition is far more accurate that the **64**-bit FP approximation of the machine. The third line prints the first **32** digits of the real value of **Pi** that I got from:

https://www.wolframalpha.com/

On a computer with **64**-bit double precision FP, the program's output looks like this:

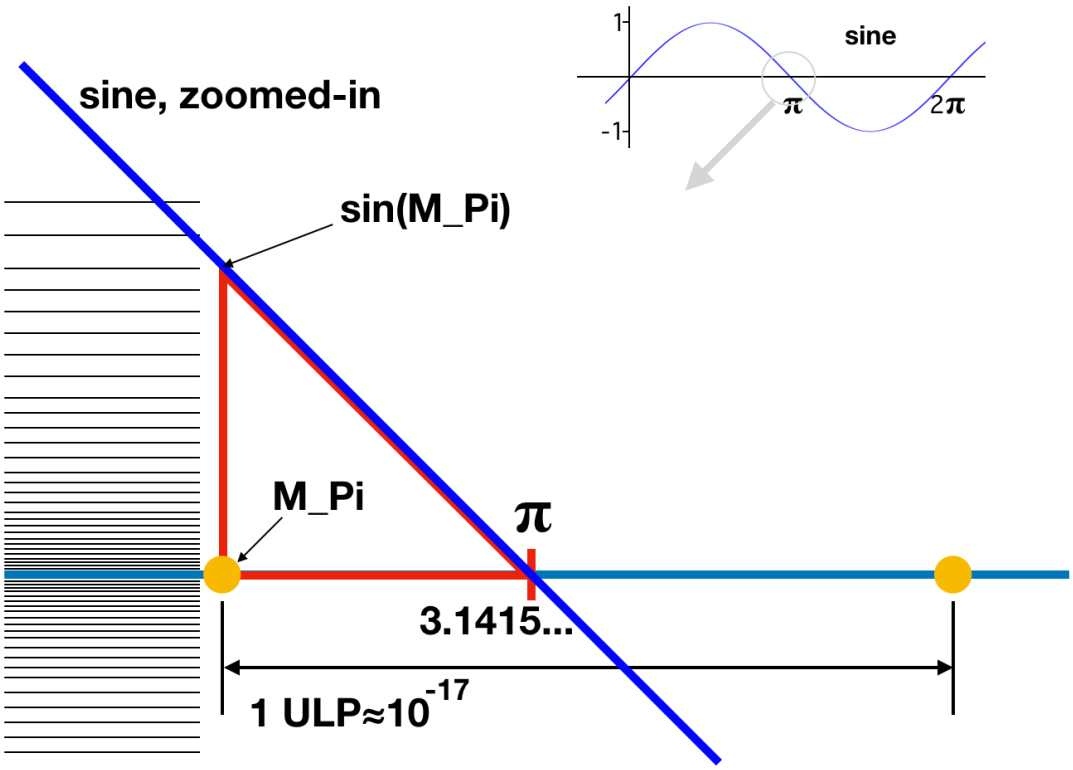
```
M_Pi      : 3.14159265358979311599796346854420
sin(M_Pi): 0.00000000000000012246467991473532
Real Pi   : 3.14159265358979323846264338327795...
```

Comparing the first and the third lines reveals that the FP representation is accurate up to the digits "**...9793**", or **15** digits after the dot. That is about the limit for a **64**-bit FP representation. What about the other digits after **9793**? Remember that this is the base-**10** representation of what is a base-**2** value in the computer. So the additional **16** bits are not random: they are accurately representing the **64**-bit base-**2** number for **M_Pi**.

The second line is not zero, because **M_Pi** is not exactly **Pi**. Thanks to the wonders of FP arithmetic, it is a number very close to zero. Now add the first two lines and compare to

the third: surprise! An exact match. So with this almost trivial sum you can get your computer to produce the first **32** digits of **Pi**, while its FP arithmetic has a precision of just **16** digits.

One caveat: this only works if you have a really good implementation of the **sin()** function. The machine used, running a modern **64**-bit Linux OS, has that.



In the figure above an extremely zoomed-in part of the sine function is shown around the X-axis value of **Pi**. The two yellow dots are the nearest points on the number line that are representable by **64**-bit floating point values. They are **1** ULP apart, which at this point on the X-axis, is about $\mathbf{10^{-17}}$.

The explanation of why the **32**-digit addition works has two ingredients. The first ingredient is simple maths: close to **Pi**, the sine function is almost equal to a straight line at an angle of **45** degrees, see figure. So the distance from a point on the number line to **Pi** in the X-direction is equal to the distance to the straight line in the Y-direction, see the even-sided red triangle. This vertical distance is **sin(M_Pi)**, which we have added to **M_Pi** in the X direction. And that got us **16** digits closer to real **Pi**.

The second ingredient is more tricky: it requires the understanding that the density of the floating point field is much higher along the Y-axis than on the X-axis. In the figure, this is represented (poorly) by the density of horizontal lines on the left side. In reality, the lines are much closer together. On the X-axis around **Pi**, the value of **1** ULP is in the order of $\mathbf{10^{-17}}$. On the Y-axis, closest to zero, **1** ULP is in the order of a mind-bogglingly small $\mathbf{10^{-300}}$. Around **sin(M_PI)**, the density is much lower, but **1** ULP there is still about

10^{-33}. So, the trick is that near zero, the value of **sin(M_PI)** can be approximated **16** digits more precisely than the value of **M_PI** itself.

The Third Lesson: Cosine is Crazy Accurate for Large x

My third discovery came when writing tests for the **cos()** function in the C-library. I found that the results of the tests were much more accurate than I had anticipated to be possible.

A reasonable expectation is that the computed value of **cos(x)** is only accurate to about **1** ULP in the range from **-2 Pi** to **Pi**. This is about as good as it gets because **1** ULP means that only the last bit is not accurate due to the rounding from the real value to the closest FP-dot on the number line.

A surprise came when I looked at the accuracy of **cos(x)** for a very large value of **x**: **10** to the power **100**. It turns out that the cosine function is accurate to about **1** ULP also for that value! Trying some more big values revealed that that was no coincidence: their computed cosine values were all eerily accurate.

Like sine, the cosine function is a periodic function that repeats itself every **2 Pi**. This means that **cos(x) = cos (n*2*Pi + x)** for all whole numbers n. The cosine library implementation uses this property so that it only needs an approximation function that is accurate between **0** and **2 Pi**. All values outside of the range **0** to **2 Pi** are mapped into that range by computing the modulo of **x** divided by **2 Pi**. But here lies the problem. To compute an accurate modulo value of a hundred digit number, you need a value of **Pi** that is accurate to hundred digits. As we have seen before, the FP value **M_Pi** has only about **16** accurate digits. With that precision, the error would have a magnitude of **10** to the power **84**: more than enough to make the outcome about random.

In order to explain how the cosine of a hundred digit number can still be so accurate, let's take a look at the implementation. Turns out we have to thank the clever engineers of Sun Microsystem for this, and their bosses for making the library open source. The so called *range reduction* part of the cosine function adapts itself to the size (magnitude) of the value of **x**. The implementation indeed knows almost **500** digits of **Pi**, and will use them when necessary. These computations are not simply done in **64**-bit double arithmetic, but use successive refinement to get the modulo right. You can look at the sources here:

https://github.com/bminor/newlib/blob/master/newlib/libm/math/e_rem_pio2.c

The Fourth Lesson: Absolute Versus Relative Error

So now we do have amazingly accurate FP results for **cos(x)** for very large values of **x**. But is that accuracy useful? I argue it is not, and will even state that I think that that accuracy can be harmful.

Here is my reasoning: by the time you work with numbers as large as **10** to the power **100**, the FP dots on the number line will be very sparse. Of those hundred digits, only

the first sixteen or so are accurate. This means that the space between FP-dots on the number line in this region is in the order of **10** to the power **84**. So in this region, the size of a **1** ULP error is **10** to the power **84**. That is huge, even compared to the number of atoms in the observable universe (**10** to the power **82** by some estimates). And in that empty space between dots, the cosine function makes a lot of waves. That means that when you are computing with values in this region, a relatively 'tiny' **1** ULP error in the input **x** is going to result in a completely random value of **cos(x)**. It does not matter that that value is accurate to **16** decimal digits -- if the input is rounded by even the tiniest amount, **cos(x)** is going to produce **16** accurate but totally useless digits because the other **84** digits of the input are incorrect.

Here is a more mundane example:

```
#include <stdio.h>
#include <math.h>
#define M_Pi    3.14159265358979323846264338327795
int main (void) {
    printf ("%.16g\n", sin (355.0));
    printf ("%.16g\n", sin (355.0 + 2*M_Pi));
    return 0;
}
```

Applying the math identity **sin (x) = sin (x + 2Pi)** would make you believe that the output from the two print statements should be very close to equal. Right...?

.....Wrong. Here is what it prints on my machine:

```
-3.014435335948845e-05
-3.014435333792724e-05
          ^
```

The two already differ in the **10**th decimal. That seems like a big difference given that the **sin(355)** is computed accurately to **16** digits, **355+2*M_Pi** is computed accurately to **16** digits, and **sin(355+2*M_Pi)** is computed accurately to **16** digits. So why are the results different already after **10** decimals?

The reason is that **sin(x)** is very sensitive to the accuracy of its input when its result value is close to zero. Not by accident, with **x=355** this is the case. (Note that **355/113** is a fair approximation of **Pi** to six accurate digits.) As before, around a result value of zero, the sine function is close to diagonal. This means that the absolute value of an error in its input will result in a similar absolute error value in its output. The input is accurate to about **16** digits. With **355**, that is about **13** digits after the decimal point. Let's put that together with the result value (E for inaccurate digits):

```
355.0000000000000EEEEEE
  0.0000301443533EEEEEE (negative)
```

The E-digits match up precisely. Similarly, the input value **355+2*M_Pi** is first rounded to the nearest dot on the FP number line before it is passed to the sine function. Thus, it does not have more than **13** accurate digits after the decimal point, and the result value

also has no more than **13** accurate digits after the decimal point. Around **355**, the dots on FP number line are just too sparse to more accurately represent the input values.

Corollary: when working with the sine function in a computer program, the result will not be more accurate that the absolute error in its input value. For input value larger than **10** to the power **16**, the absolute error is in the order of **1** and more, and the sine function is useless for practical purposes. (Yet still eerily accurate...)

So How Does This Relate to SuperTest?

As stated in the beginning, this quest began with making improvements to SuperTest's Math testing. And so we did. The improved math library testing is now part of SuperTest, accurate to the last ULP if that is the standard you want to compare to. But your library does not have to be that accurate. The C language specification does not put accuracy requirements on the math library. In an embedded application, you may not want and may not need a cosine that computes with **500** digits of **Pi**. It may be too slow or too large. This paper should help with defining reasonable bounds for the accuracy that your application requires.