A Tale of Two Numbers

With the Pentium, there is a very small chance of making a very large error

by Cleve Moler

This is the tale of two numbers, and how they found their way over the Internet to the front pages of the world's newspapers on Thanksgiving Day, embarrassing the world's premier computer chip manufacturer. The first number is the 12-digit integer

$$p = 824633702441$$

Thomas Nicely of Lynchburg College in Virginia is interested in this number because both p and $p+2$ are prime. Two consecutive odd numbers that are both prime are called *twin primes*. Nicely's research involves the distribution of twin primes and, in particular, the sum of their reciprocals,

$$S = 1/5 + 1/7 + ... + 1/29 + 1/31 + ... + 1/p + 1/(p+2) + \ldots$$

It is known that this infinite sum has a finite value, but nobody knows what that value is. For over a year, Nicely has been carrying out various computations involving twin, triple, and quadruple primes. One of his objectives has been to demonstrate that today's PCs are just as effective as supercomputers for this kind of research. He was using half a dozen different computers in his work until March, when he added a machine based on Intel Corporation's Pentium chip to his collection.

"In June, Nicely noticed inconsistencies among several different quantities he was computing. They were accurate to only roughly single precision, even though he was using double precision."

In June, Nicely noticed inconsistencies among several different quantities he was computing. He traced them to the values he was getting for $1/p$ and $1/(p+2)$. Although he was using double precision, those values were accurate only to roughly single precision. He placed the blame at first on his own programs, then on the compiler and operating system, then on the bus and motherboard in his machine. By late October though, he was convinced that the difficulty was in the chip itself.

On October 30, in an email message to several other people who had access to Pentium systems, Nicely said that he thought there was a bug in the processor's floating-point arithmetic unit. One of the recipients of Nicely's memo posted it on the CompuServe network. Alex Wolfe, a reporter for *EE Times*, spotted the posting and forwarded it to Terje Mathisen.

Terje Mathisen, a computer jock at Norsk Hydro in Oslo, Norway, has published articles on programming the Pentium, and has posted notes on the Internet about the accuracy of its transcendental functions. Within hours of receiving Wolfe's query, Mathisen confirmed Nicely's example, wrote a short, assembly-language test program, and on November 3, initiated a chain of Internet postings in the newsgroup *comp.sys.intel* about the FDIV bug. (FDIV is the floating-point divide instruction on the Pentium.) A day later, Andreas Kaiser of Germany posted a list of two dozen numbers whose reciprocals are computed to only single-precision accuracy on the Pentium.

On November 7, the *EE Times* published the first news article on the bug. The headline on the front page story by Wolfe was "Intel Fixes a Pentium FPU Glitch".

Tim Coe is an engineer at Vitesse Semiconductor in Southern California. He has designed floating-point arithmetic units himself and saw in Kaiser's list of erroneous reciprocals clues to how the Pentium designers had tackled the same task. He surmised, correctly it turns out, that the Pentium's division instruction employs a technique known to chip designers as a radix-4 SRT algorithm. This produces two bits of quotient per machine cycle and thereby makes the Pentium twice as fast at division as previous Intel chips running at the same clock rate.

Coe created a model that explained all the errors Kaiser had reported in reciprocals. He also realized that the relative errors were potentially even larger for division operations involving numerators other than 1. His model led him to a pair of seven-digit integers whose ratio,

$$c = 4195835 / 3145727$$

appeared to be an instance of the worst-case error. Coe did his analysis without actually using a Pentium—he doesn't own one. To verify his prediction, he had to bundle his year-and-a-half-old daughter into his car, drive to a local computer store, and borrow a demonstration machine.

The true value of Coe's ratio is

$$c = 1.33382044 \ldots.$$

But computed on a Pentium, it is

$$c = 1.33373906 \ldots.$$

The computed quotient is accurate to only 14 bits. The relative error is $6.1 \cdot 10^{-5}$. This is worse than single-precision roundoff error and more than ten orders of magnitude worse than what we expect from double-precision computation. The error doesn't occur very often, but when it does, it can be very significant. We are faced with a very small probability of making a very big error.

I first learned of the FDIV bug a few days before Coe's revelation, from an electronic mailing list maintained by David Hough; at that point I began to follow *comp.sys.intel*. At first, I was curious but not alarmed. It seemed to be some kind of single- versus double-precision problem, which, although annoying, is not all that uncommon. But Coe's discovery, together with a few customer calls to The MathWorks tech support department, raised my level of interest considerably.

On November 15, I posted a summary of what I knew then to both the Intel and the MATLAB newsgroups, using Nicely's prime and Coe's ratio as examples. I also pointed out that the divisors in both cases are a little less than three times a power of two:

$$824633702441 = 3 \cdot 2^{38} - 18391$$

and

$$3145727 = 3 \cdot 2^{20} - 1$$

By this time, the Net had become hyperactive, and my posting was redistributed widely. A week later, reporters for major newspapers and news services had photocopies of faxed copies of printouts of my posting.

The story began to get popular press coverage when Net activists called their local newspapers and TV stations. I was interviewed by CNN on Tuesday, November 22, and spent most of the next day on the telephone with other reporters. My picture ran in *The New York Times* and *The San Jose Mercury News* on Thursday, which happened to be Thanksgiving. The *Times* story included a sidebar, titled "Close, But Not Close Enough," which used Coe's ratio to illustrate the problem. The story was also front-page news in the *Boston Globe*, with the headline "Sorry, Wrong Number." The *Globe*'s sidebar demonstrated the error by evaluating Coe's ratio in a spreadsheet.

In the month since I learned of Nicely's prime and Coe's ratio they have certainly earned their place in the Great Numbers Hall of Fame.

One challenging aspect of all this has been explaining to the press just how big the error is. The focus of attention has been on what we numerical analysts call the residual,

$$r = 4195835 - (4195835 / 3145727)(3145727)$$

With exact computation, r would be zero, but on the Pentium it is 256. To most people, 256 seems like a pretty big error. But when compared with the input data, of course, it doesn't seem so large. This gets us into relative error and significant digits—terms that are not encountered in everyday journalistic prose. There was even confusion on the part of some reporters about where to start counting "decimal digits." Not everybody got the details right. *The New York Times* was the only publication I saw that thought to show the actual value of the erroneous quotient. The British publication, *The Economist*, had the good sense to describe the error as 0.006%. In retrospect, perhaps the best description of the error would have been "about 61 parts per million."

Since double-precision floating-point computation is vital to MATLAB, and since Pentium-based machines account for a significant portion of new MATLAB usage, we decided early on to provide a release of our software that works around the bug. My first thought was to modify our source code so that every division operation

```
z = x/y
```

was followed by a residual calculation

```
r = x - y*z
```

When the residual was too large, the division could be redone in software using Newton's method. We have now abandoned both aspects of this approach—we don't compute the residual, and we don't do software emulation.

"Since double- precision floating- point computation is vital to MATLAB, and since Pentium-based machines account for a significant portion of new MATLAB usage, we decided early on to provide a release of our software which works around the bug."

Coe, Mathisen, and I are now working with Peter Tang of Argonne National Laboratory (on sabbatical in Hong Kong) and a group of hardware and software engineers from Intel, who are in California and Oregon. We have never met face-to-face. Our collaboration is all by email and telephone (11 a.m. in Massachusetts is 8 a.m. in California and Oregon, 5 p.m. in Oslo, and 1 a.m. in Hong Kong). We are developing, implementing, testing, and proving the correctness of software that will work around the FDIV bug and related bugs in the Pentium's on-chip tangent, arctangent, and remainder instructions. We will offer the software to compiler vendors, commercial software developers, and individuals via the Net. We hope this software will be used by anyone who is doing floating-point arithmetic on a Pentium and is unable to replace the chip.

The Pentium's division woes can be traced to five missing entries in a lookup table that is actually part of the chip's circuitry. The radix-4 SRT algorithm is a variation of the familiar long division technique we all learned in school. Each step of the algorithm takes one machine cycle and appends another two-bit digit to the quotient. Both the quotient and the remainder are represented in a redundant fashion as the difference between two numbers. In this way, the next digit of the quotient can be obtained by table lookup, with several high-order bits from both the divisor and the remainder used as indices into the table. The table is not rectangular; a triangular portion of it is eliminated by constraints on the possible indices.

When the algorithm was implemented in silicon, five entries on the boundary of this triangular portion were dropped. These missing entries (each of which should have a value of +2) effectively mean that, for certain combinations of bits developed during the division process, the chip makes an error while updating the remainder.

The key to our workaround is the fact that the chip does a perfectly good job with division almost all the time. It is simply a question of avoiding the operands that do not work very well, which our software accomplishes with a quick test of the divisor before each attempted FDIV. The absence of dangerous bit patterns is an indication that the FDIV can be done safely. The presence of one of the patterns is not a guarantee that the error will occur, just a signal that it might. In this case, scaling both numerator and denominator by 15/16 takes the divisor out of the unsafe region and ensures that the subsequent division will be fully accurate.

Pentium-Safe Division

The algorithm is:

```
function z = (x,y)
if at_risk(y)
   x = (15/16)*x;
   y = (15/16)*y;
end
z = x/y;
```

The safety filter is:

```
function a = at_risk(y)
f = and(hex(y),'000FF00000000000');
a = any(f == ['1F'; '4F'; '7F'; 'AF'; 'DF'])
```

With this approach, it is not necessary to test the magnitude of the residual resulting from a division. It is known *a priori* that all divisions will produce fully accurate results. If desired, an additional test can compare the result of scaled and unscaled divisions and count the number of errors that would occur on an unmodified Pentium. We will offer this test in MATLAB, but it can be turned off for maximum execution speed.

The regions to be avoided can be characterized by examining the high-order bits of the fraction in the IEEE floating-point representation. Floating-point numbers in any of the three available precisions can be thought of as real numbers of the form

$$\pm(1+f)\cdot 2^e$$

where e is an integer in the range determined by under- and overflow, and f is a rational number in the half-open interval [0,1). The eight high-order bits of f divide this interval into $2^8 = 256$ subintervals. Five of these subintervals contain all the at-risk divisors. The hexadecimal representations of the leading bits of the five subintervals are

1F, 4F, 7F, AF, and DF

(As I write this article, Coe, Tang, and some of the Intel people are still working on a proof of correctness. The ultimate version of the algorithm may well use a pattern with more than eight bits, but the idea will be the same.)

A picture of the floating-point numbers between any two powers-of-two looks something like this:

```
[____()______()______()______()______()____]
      1F      4F      7F      AF      DF
```

The length of each of the five subintervals is 1/256 of the overall length. The result of any division involving a divisor outside these subintervals will be accurate. Only a small fraction of divisions involving divisors inside the subintervals produce erroneous results, but the easiest, and quickest, approach is to avoid the subintervals altogether. Scaling the numerator and denominator by 15/16 shifts them to a safe region. (I had originally proposed scaling by 3/4, but this takes the 7F subinterval into the 1F subinterval.)

For example, the denominator in Coe's ratio is

$$3145727 = 1.49999952316284\cdot 2^{21}$$

The hexadecimal representation is printed as 4147FFFF80000000. So $e = 21$, $f = 2^{-1} - 2^{-21}$, and the number is close to the right end point of the 7F subinterval. The 17 consecutive high-order 1 bits in f conspire with the bits in Coe's numerator to produce an instance of worst-case error.

The Pentium, like most other microprocessors, saves floating-point numbers in memory in either a 32-bit single-precision or a 64-bit double-precision format. In the floating-point arithmetic unit itself, however, an 80-bit extended-precision format is used for the calculations. Thus, if an at-risk denominator is encountered in either single or double precision, it can be scaled by 15/16 and extended precision can be used for the division. Then, when the resulting quotient is rounded back to the working precision, it will yield exactly the same result, down to the last bit, that would be produced by a chip without the bug.

The entire algorithm can be summarized in the few lines of MATLAB code shown in the box. This is actually MATLAB pseudocode, because MATLAB does not do bit-level operations on floating-point numbers. Moreover, we do not make the distinction between the 64- and 80-bit representations that ensure full accuracy when scaling is required. But I hope this explanation has given readers an idea of the thinking that went into the workaround. ■

Cleve Moler is chairman and co-founder of The MathWorks. His email address is moler@mathworks.com