<BACK

by Jack W. Crenshaw

Pyramids: The Last Word

Well, it seems I really started something with my columns about the Great Pyramid and the significance of its geometry. I noted, as several Egyptologists have, that the ratio of height to base of the Great Pyramid is $\pi/2$ to considerable accuracy. I also mentioned the Golden Ratio:

$\phi = 1.618 \ldots$

which is often claimed to give the most pleasing proportions of a rectangle. The dimensions of the Pyramid and the things inside it indicate that the builders knew both π and ϕ, and also the Pythagorean theorem, things that orthodox mathematics tells us weren't known until thousands of years later.

These ancient mysteries struck sensitive chords in people of all stripes, and about half of them sent me e-mail. The responses varied from readers of *Skeptical Enquirer* to *National Enquirer* and from mildly interested readers to folks who offered me rides in their pals' flying saucers to see the face on Mars. The more credulous writers see the suggestion that the Egyptians knew things they weren't supposed to know as proof that they built the pyramids using antigravity sky-cranes imported from Atlantis. The more skeptical not only see the proportions of the Pyramid as mere coincidence, they interpret discussion of the subject as a superstitious evil that needs to be stamped out in the name of Science.

Among the more skeptical is Eric Krieg, an avid *Skeptical Enquirer* reader and an amateur debunker of occult claims. Krieg writes, "Please consider a conventional explanation to an extraordinary myth." Krieg's concern is that, "The Great Pyramid and related myths have been the focal point of a large number of cults and occultic thinking."

At this point, the worth of the Golden Ratio is moot. Artists and architects think it's pleasing, so they design things using it.

That's true, but I don't think you can blame the Pyramid for that, though you might blame the builders. Rightly or wrongly, people have not always published their knowledge freely, for the good of humankind, but have sometimes kept it to themselves. Renaissance mathematicians often coded their discoveries in anagrams or used similar devices to enshrine their knowledge without revealing it. People have even constructed religions and cults around esoteric knowledge; witness the Pythagoreans of ancient Greece, the Masons of the last 300 years, and the present-day Rosicrucians. If you doubt that the Pyramid and its hints of secret knowledge get into strange places, take another look at the back side of any dollar bill.

As a matter of fact, it seems to me that the cultic aspects strengthen rather than weaken the case for the Great Pyramid as a monument to this knowledge. As Krieg rightly points out, "There are much more reasonable ways to represent a number than to build a stone pyramid." No reasonable culture would expend the kind of energy the Egyptians did, encoding their knowledge into a monument 400 feet high, but a cult based on esoteric knowledge with the power to direct a great labor force just might.

NO AESTHETIC PREFERENCE

Krieg doesn't believe the Egyptians knew the Golden Ratio and claims there is no proof that the ratio is more aesthetically pleasing than any other. He references a test by computer scientist George Markowsky published in the January 1992 *College Mathematics Journal*. In double-blind tests, according to Krieg, "People have no particular aesthetic preference for the Golden Rectangle."

I frankly admit to being surprised. I thought that, while the source of the Golden Ratio might be in doubt, its existence and value was surely not. To my knowledge, this ratio is taught in every architectural school and art school in the world and has been for centuries. It's in the dictionary and the encyclopedia. Durer, da Vinci, Palladio, Signac, and Seurat all used it widely. Seurat's "La Parade" shows not one but five such proportions.

To find out for sure, I went where I always go when seeking knowledge: to CompuServe's Software Development forum. Several members confirmed my belief that the Golden Ratio is used in art and architecture. They also pointed out items in everyday life that more or less conform to this ratio, including playing cards, 3-by-5-inch index cards, 5-by-8-inch photos, business cards, credit cards, Post-It notes, and so on.

At this point, the worth of the Golden Ratio is moot. Artists and architects think it's pleasing, so they design things using it. Whether people find its proportions pleasing, as da Vinci thought, or not, as Markowsky's test hints, doesn't matter anymore.

To appease Krieg and other skeptics, it's only fair to acknowledge the great controversy over the meaning of the proportions of the Great Pyramid. Orthodox archeologists and orthodox mathematicians do not, repeat *not*, accept that the Pyramid builders knew π, ϕ, or the Pythagorean theorem. On the other hand, it's hard to argue with a million-ton counter-example. You don't have to believe in UFOs or the occult to accept the fact that people of ancient times could be at least as smart as those of today, that mathematical discoveries, like geographical ones, might have been made more than once and lost again, or that people might use their knowledge for less than rational purposes.

We've spent enough time and space on this subject. We will speak of it no more.

BACK TO FIXED POINT

Instead, let's get back to the point, which is fixed-point arithmetic. Last month, I showed you how to add, subtract, and multiply fixed-point numbers. As you may recall, we adopted a notation Bn, where n stands for the number of bits after the binary point. The default format, B0, implies a pure fraction in which all bits except the sign bit follow the decimal point. In other words, the number:

0x4000, B0

is equivalent to 0.5. The number:

0x7fff, B0

is just under one—0.999969482, to be precise. Decreasing n decreases the size of the number, and increasing n increases it. If we're using 16-bit integer numbers, a number with format B15 has the decimal point all the way to the right, so it's a pure integer.

Fixed-point format is a bit like floating point in that every number carries with it an exponent. When we perform arithmetic on such numbers, we must deal with the exponents. The difference is that in floating-point arithmetic, the exponents are adjusted as part of the process. In fixed-point arithmetic, the exponent is, well, fixed. You can't alter it, but you still must deal with it.

We've learned that when we multiply two fixed-point numbers, we can always plan on shifting the result.

Last month, you learned how to add, subtract, and multiply fixed-point numbers. This month, we'll look at the remaining operation: division.

DIVISION

We learned that when we multiply two fixed-point numbers, we can always plan on shifting the result. The reason bears repeating, because it influences the division operation as well.

When we perform fixed-point arithmetic using integers on a modern microprocessor, we really must deal with two factors. First, the CPU is a pure integer machine. It doesn't know anything about fixed point. To the CPU, 0x4000 is 16384, not 0.5 or any other decimal fraction. When we multiply two 15-bit integers, the CPU reasonably assumes that we're going to get a 32-bit result, and all modern microprocessors are designed to support "int-to-long" multiplication. That's all well and good if we're truly multiplying integers, but when we're using fixed-point arithmetic, the answer we want, that is, the most significant 16 bits, is always in the upper half of the product. Because of that, we learned that we must shift the product right 15 bits (not 16) to get back to a B0 result.

The second factor is the programming language we're using. Those programmers who use assembly language take a sharp right turn from those of us who use C or C++. The reason is that, while the CPU understands that multiplying gives a 32-bit result, the language does not. In C, any arithmetic operation on two `int`s, including multiplication and division, yields a result that's also an `int`. Since this isn't what we want, we must cast one or both operands to a long before the multiplication. (Afterwards is too late!) The standard fixed-point multiplication in C looks like:

```
z = ((long)x * y) >> 15;
```

It's a bother to have to write this code for every single product, which is why I claim that C is poorly suited to real-time programming. We'll get back to that point in a moment.

We must deal with similar issues in the case of division, and again, C and assembly language programmers must take different forks in the road. Assembly language is complicated by the fact that, just as the designers of CPUs have assumed that multiplying any two 16-bit integers gives a 32-bit result, they have also assumed that dividing a 32-bit number by a 16-bit one gives a 16-bit result.

This is a false assumption. To prove it, you need only divide any long number by unity. The result is the same long number, which will not fit into 16 bits. Why the CPU designers have historically done things this way is beyond me, because it takes very little extra logic to leave a 32-bit quotient, but it is a fact we must live with.

Going to fixed-point arithmetic helps a little, but not much. Just as we shifted the product right 15 bits after multiplying, we must shift left 15 bits before dividing. However, even that step doesn't necessarily keep us out of trouble. Take a look at the following quotient:

0x7fff, B0 (0.999969)
0x4000, B0 (0.500000)

The quotient is 1.999938965, which is too large to get back into a B0 scaling. We must shift the result back one more bit to the right, and scale it to B1. To

avoid the post-division shift, we can shift the original divisor only 14 bits instead of 16:

divisor:	0x7fff
shift left 14:	0x1fffc000
divide:	0x7fff, B1

The reason we had to adjust the scale was because the divisor 0.5 was smaller than the dividend 0.999969. Recognize, though, that the resulting scale of B-1 is only valid for the particular combination of operands we used. Suppose the divisor were 0x0001:

0x7fff, B0 (0.999969)

divided by:

0x0001, B0 (0.000031)

equals:

0x7fff, B15 (32767.00)

All the input operands were scaled B0, but results in the two cases are radically different. It may seem to make no sense that the scale of the result should be determined by the scale of the input operands. If you feel this way, you're thinking in terms of floating point, where the exponent has a true relationship with the number. The scales we're using here really have nothing whatever to do with the values that may be in them at any given time; they're static scalings rather than the dynamic ones of floating point (which is why they're called "fixed"). And they're scalings you've chosen, however arbitrarily, to use. You can represent any number in any scaling, though the results may be nonsensical if you choose badly.

This result simply serves to emphasize the key point: In using fixed-point arithmetic, you must consider the possible range of the results of every computation and scale according to the worst case. Say that over to yourself 100 times or until you get the message.

Fortunately, when we program in C and C++, the operand promotion rules work in our favor for a change. That is, while the CPU designers may think that the quotient of a 32-bit number by a 16-bit number should yield a 16-bit result, C knows better and assumes that the result of a long divided by an int is another long. This saves us one minor bother but still doesn't relieve us of the more serious one, shifting before the division. The code to perform the two division examples is:

```
z = ((long)x << 14) / y;
  // result scaled B1
z = (long)x / y;
  // result scaled B15
```

Note carefully the sequence of things. You must promote the dividend before the shift, not after, and perform the division after the shift, not before.

From the two examples, you should be able to see the general rule: Shift left 15 bits minus the number that the scaling must change. In the case where both the operands have nonzero scales,

subtract the scale factors. So if we're dividing a number scaled Bn by a number scaled Bm and want the result to be scaled Br, the shift factor is:

s = 15 + n - m - r

A BETTER WAY

Writing fixed-point software is no fun; it's very tricky stuff. You must compute the best scaling for every variable in the program, including all temporaries computed as part of a formula. It is not an easy job. And C and C++ don't give us much help. The plain facts are, we are computing with both a CPU and a programming language that don't understand fixed-point, so we are making do, performing fixed-point arithmetic with integer tools that include a brain-damaged division algorithm.

It doesn't have to be that way. We should be able to ask for help in both areas. Twenty years ago, most aerospace manufacturers had their own proprietary flight computer. Many of them had fixed-point arithmetic built in. The Honeywell computer, for example, performed both multiplication and division directly using B0 format and gave the result without shifting. We programmers only had to deal with shifting to get scales other than B0. All those proprietary processors were done in by the Department of Defense's Military Computer Family (MCF) initiative, an effort to standardize on an instruction set architecture. This effort led to the MIL-STD-1750A processor, which most folks now concede is obsolete.

Two programming languages, Jovial and Ada, were developed specifically for writing real-time programs, again mostly for the military. Both languages include fixed-point arithmetic as part of their specification. Jovial is also obsolete; it has been ruled unsuitable for new defense projects. Its implementation of fixed point was never very efficient, anyhow.

Additionally, the designers of Ada tried to be just a little too cute. You and I know that we'll probably never see another computer that doesn't have a word length a power of two bits long:

Floating point eliminates the dangers of overflow or poor accuracy.

8, 16, 32, or 64 bits. The Ada designers, however, having seen too many languages designed with a given computer in mind (C and PDP-11, for example) didn't want to tie the language to any specific word size. In Ada, instead of specifying a fixed-point word length, you specify its range and precision and let the compiler decide upon a scaling. Theoretically, the compiler could choose any word length, though presumably, the algorithm making that decision would favor words that the target CPU could actually use.

What happened next is a bit obscure. It could be that the compiler vendors, under pressure to get the compiler delivered, took shortcuts in the area of fixed point. After all, the language specification says that Ada should support fixed point, but it doesn't say that it has to generate efficient object code. The explanation could simply be that language designers weren't mathematicians. Whatever the reason, the early Ada compilers generated terrible code when fixed point was used—on the order of 10 times slower than other languages. The situation was so bad that most Ada shops had a standing rule: Thou shalt not use fixed point.

I'm sure the code generation was improved in the later Ada compilers, but fixed-point arithmetic still seems to rank way down the priority list when it comes to compiler improvements.

So here we are in 1995, 20 years after the MCF effort began, with computers that don't understand fixed-point arithmetic and languages that either don't understand it, are rarely used, or generate poor code. The chances that this situation will change in the near future are slim, because we're likely to get fast floating-point processors that make the whole problem go away. Floating point is a much better solution anyway, since it eliminates the dangers of overflow or poor accuracy inherent in the use of fixed-point arithmetic. Pray for floating point. In the meantime, we make do, writing strange-looking code and carefully scaling every arithmetic value. Careful design and exhaustive testing over the whole range of expected inputs is essential.

A side note to employers: Whatever you're paying your programmers who are doing this stuff, it's not enough. Programming fixed-point arithmetic requires the most exacting and careful design and testing, which is the kind of thing they don't teach in college. If you decide to cut corners on your staffing and hire a programmer whose last job was writing an Accounts Receivable program in Basic, you deserve whatever you get.

OPPORTUNITY KNOCKS

When I was developing navigation and guidance software, I spent most of my time making sure I got the scaling right. Testing was not easy, mainly because we didn't have very good tools—usually only a hex debugger. Like all of my ilk, I got pretty good at converting hex numbers into fixed-point numbers. On a good day, I could keep the calculator just under the boiling point. But when you stop and think about it, paying someone big bucks to run a pocket calculator is a pretty inefficient way to do business. And aside from the conversions themselves, there's also the fact that almost every number is scaled to a different format. So you not only need a calculator close at hand but also a list of all your program variables and their scales. There ought to be a better way.

I always wished I could take the time to sit down and write a program to help me remember what the scalings were for each variable and help me perform the conversions. The idea goes something like this: When I was in the

design phase, I'd input the variables that the program needed and the expected range and precision values, similar to the Ada representation. The program would suggest a scale factor and word length for me, which I could accept or modify. (Sometimes, remember, we can avoid a shift by relaxing the precision requirement a bit.) When a new variable depended on an operation on two existing variables, I'd want to be able to tell the program the dependence and let it compute the resulting range and scale.

After the design process was complete, I'd want to be able to save the information in a file. During the testing phase, I could enter the observed hex value for any variable. The program, drawing on its database, would know how to convert the data to decimal and display it for me. I could also enter data in decimal and have the program convert it to hex, with the right scaling.

What I just described is a symbolic debugger, but one that understands fixed point. Most symbolic debuggers can display data in virtually any format, including complicated structures. This debugger would add just one more layer, which would be the conversion of fixed-point numbers.

Looking back on it, I realize that my original goals were limited. A tool that helps me design my scaling and decipher it later would be great, and I wouldn't slam the door on anyone offering me one. But if I'm going to let a computer program design the scaling for me, why not take it that one last step and have it generate the code as well?

Now we're talking about a code generator instead of a debugging tool. More precisely, we're talking about a new language, an extension to C and C++ complete with a preprocessor (shades of C++ and cfront) and a symbolic debugger.

Wouldn't that be great? Would you use it? I sure would. So here's the deal: Write yourself such a preprocessor/-debugger system and get rich. I'll be your first customer.

I intended to finish up with a practical example, the sine function implemented in fixed point. As usual, my page-count meter runneth over, so I'll have to defer the example until next month. Perhaps that's better, anyway, because I want to take time to look at lots of little refinements. If we're going to do such a function, I want it to be one you'll really be able to use and never have to rewrite again. That takes a little more effort than just a simple example, so it's a good thing to put off until we can do it justice. ESP

Jack W. Crenshaw is a staff scientist at Invivo Research in Orlando, FL. He did much early work in the space program and has developed numerous analysis and real-time programs. He holds a PhD in physics from Auburn University. Crenshaw enjoys contact and can be reached via e-mail at 72325.1327@compuserve.com.