The Stata Journal

The *Stata Journal*, electronic version (ISSN 1536-8734) is a publication of Stata Press. Stata and Mata are registered trademarks of StataCorp LP.

The Stata Journal (2006)
6, Number 4, pp. 550–560

Mata Matters: Precision

William Gould
StataCorp
College Station, TX
wgould@stata.com

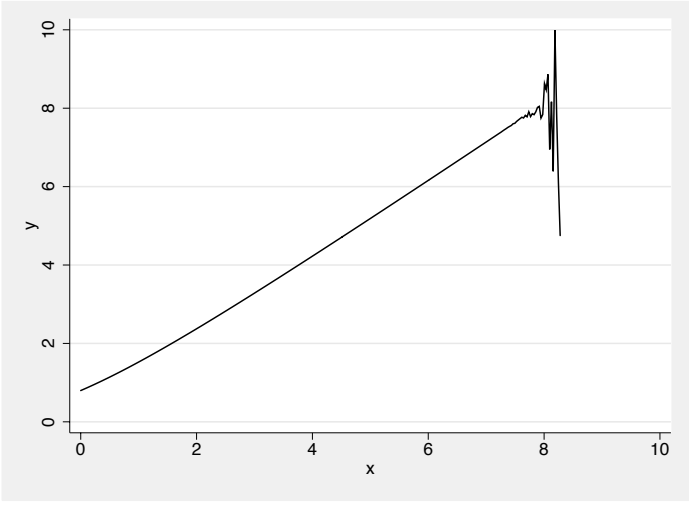
Abstract. Mata is Stata's matrix language. The Mata Matters column shows how Mata can be used interactively to solve problems and as a programming language to add new features to Stata. In this quarter's column, we look at the programming implications of the floating-point, base-2 encoding that modern computers use.

Keywords: pr0025, Mata, floating point, binary, hexadecimal

Introduction

This quarter's column is not specific to Mata. We are going to discuss how computers, and thereby Stata and Mata, store numbers such as 1, −1, 12.32, and 2.2026466e+04. You know that modern computers are binary, but did you know that some earlier computers such as the IBM 1620—a 1960s scientific computer—were decimal? There was a reason they did that. What seems an unimportant detail—how computers store numbers—has important implications for how programs and even simple expressions are written. For instance, look at this graph of Mills' ratio:

```
. graph twoway function y = normalden(x)/(1-normal(x)), range(0 9)
```



For your information, Mills' ratio is in fact nearly linear for large values of x. Do you see how the calculation went wrong? You may be thinking that the user did the calculation in single precision. That is not the cause; the above calculation was done in double

precision throughout and by highly accurate routines. The problem has to do with the very way computers store and manipulate floating-point numbers; it is inherent. The user did nothing wrong except not knowing that calculation of expressions such as `1-normal(x)` is prone to roundoff error and that the solution is to substitute `normal(-x)` and so graph `normalden(x)/normal(-x)`.

Or consider this snippet from another user's log. The user has expense data in U.S. dollars and cents, but the problem we are about to see could bite with any similar currency, meaning almost all of them.

```
. list in 1
```

	expense1	expense2
1.	3000.12	1042.58

```
. gen total = expense1 + expense2
. format total %10.0g
. list in 1
```

	expense1	expense2	total
1.	3000.12	1042.58	4042.7002

Why the error and what should the user do about it? If you are thinking single versus double precision, this time you are nearly right. The issue is not really single versus double precision, but using double precision would push the problem farther away—far enough here that the user will not even see it. Make the numbers larger, however, or add more of them, and the problem will reappear. This problem has to do with base 2 itself, and the solution is to store the dollar amounts as integers, in cents. By the way, the problem was not merely in the summing of `expense1` and `expense2`; the error occurred when the data were entered:

```
. format expense1 expense2 %10.0g
. list in 1
```

	expense1	expense2	total
1.	3000.1201	1042.58	4042.7002

If we can have problems with such simple, everyday calculations, we can imagine the problems we might have in a long program that makes hundreds or even thousands of calculations. Indeed, at this point, you are probably wondering how any calculation you have ever made got a correct result. The answer is that I did not choose these two examples at random. You can mostly ignore the implications of how computers store numbers and, with reasonable values of the input variables, you will be okay. Most programmers assume that what they don't know can't hurt them, but those programmers do not work for StataCorp. You don't want them working for you, either.

Notation

When we write a base-10 number such as 123, what we mean is $\mathbf{1}\times10^2+\mathbf{2}\times10^1+\mathbf{3}\times10^0$. Change the 10 to an 8 and we have a base-8 number: $123_8 = \mathbf{1}\times8^2+\mathbf{2}\times8^1+\mathbf{3}\times8^0$, or 83_{10}.

A number without a subscript—123, for instance—means the base-10 number, although sometimes we will write 123_{10} to emphasize the base-10 part. We will write numbers in other bases, such as 1100100_2, 1210_4, 144_8, and 64_{16}. For bases above 10, we will use letters for the extra digits. In base 12, *a* would be 10 and *b*, 11. The number $\mathrm{a2b}_{12}$ is $\mathbf{10}\times12^2+\mathbf{2}\times12^1+\mathbf{11}\times12^0$, or 1475_{10}. In base 16, we will also use *c* for 12, *d* for 13, *e* for 14, and *f* for 15, so $\mathrm{a2b}_{16} = 2603_{10}$.

We will use the period as the base point.

In base 10, we call the base point the decimal point: 123.45 means $\mathbf{1}\times10^2+\mathbf{2}\times10^1+\mathbf{3}\times10^0+\mathbf{4}\times10^{-1}+\mathbf{5}\times10^{-2}$.

In base 8, we call the base point the octal point: 123.45_8 means $\mathbf{1}\times8^2+\mathbf{2}\times8^1+\mathbf{3}\times8^0+\mathbf{4}\times8^{-1}+\mathbf{5}\times8^{-2}$, or 83.578125_{10}.

In base 2, we call the base point the binary point: 1101.01_2 means $\mathbf{1}\times2^3+\mathbf{1}\times2^2+\mathbf{0}\times2^1+\mathbf{1}\times2^0+\mathbf{0}\times2^{-1}+\mathbf{1}\times2^{-2}$, or 13.25_{10}.

In base 16, we call the base point the hexadecimal point: $\mathrm{1b.8a}_{16}$ means $\mathbf{1}\times16^1+\mathbf{11}\times16^0+\mathbf{8}\times16^{-1}+\mathbf{10}\times16^{-2}$, or 27.5390625_{10}.

A convenient trick

You know that modern computers are binary. The number 100_{10} is stored 1100100_2. The number π is stored $11.00100100001111110110101_2$, although I've omitted several digits.

Perhaps you have heard that modern computers are hexadecimal, which is to say, base 16, and perhaps you remember earlier computers that were octal. Actually, all those computers were binary, and whether we labeled them octal or hexadecimal was more a matter of notation than anything else, although the base used did indicate a fine detail of wiring.

The notation issue is this: when one base is a power of another, one can perform base conversion by parts. For instance, 4 is 2^2, so base 4 is a power of base 2, and that means a base-4 number can be converted into a base-2 number not only by the usual formulas but also by simply converting each base-4 digit independently and then writing down the results one after another.

For instance, consider the number 321_4. I can just glance at it and tell you that, in base 2, the number is 111001_2. I obtained that result by converting each digit separately. The first digit of 321_4 is 3_4, and that is 11_2. The second digit of 321_4 is 2_4, and that is 10_2. The third digit of 321_4 is 1_4, and that is 1_2, but you must write it 01_2 (each number you write down must have k digits, where k is the conversion of base -1). Now

take those results and write them one after the other. We obtained 11_2, 10_2, and 01_2, so the overall result is 111001_2.

Test it: $\mathbf{3}\times4^2 + \mathbf{2}\times4^1 + \mathbf{1}\times4^0 = \mathbf{1}\times2^5 + \mathbf{1}\times2^4 + \mathbf{1}\times2^3 + \mathbf{0}\times2^2 + \mathbf{0}\times2^1 + \mathbf{1}\times2^0 = 57$.

How to perform the conversion is more easily seen in tabular form:

```
base-4 number = | 3| 2| 1|
-------------------------
base-2 number = |11|10|01|   =  111001
```

Write down the base-4 number, leaving enough room underneath to write the equivalent binary digits. Write down the translation of each base-4 digit, and finally push the individual results together into one base-2 number.

The method can be used in the opposite direction: $111001_2 = 321_4$ because 11_2 is 3_4, 10_2 is 2_4, and 01_2 is 1_4. In tabular form:

```
base-2 number = |11|10|01|
-------------------------
base-4 number = | 3| 2| 1|   =  321
```

With this trick, you can perform base conversion by examination, and writing 321_4 is certainly more convenient than writing 111001_2. For one thing, making an error is too easy when writing in binary.

Base 8 is even more convenient. In base 8, the largest digit is 7, and that is 111_2 in binary. So we convert in three-digit groups: 111001_2 is

```
base-2 number = |111|001|
-------------------------
base-8 number = |  7|  1|    =  71
```

We can write 71_8 and, if we later need the binary representation back, we can write it down by examination: the first digit, 7_8, translates to 111_2; the second digit, 1_8, translates to 001_2; and we have 111001_2.

These days, we use base 16. To convert 111001_2 into base 16, we divide the original into four-digit groups. 111001_2 has only six digits, so we add two more zeros, on the left of course, which does not change the meaning of the number because 00111001_2 equals 111001_2.

```
base-2 number  = |0011|1001|
---------------------------
base-16 number = |   3|   9|    = 39
```

Bases 4, 8, and 16 are just more convenient ways to write base-2 numbers. We are going to use these other bases in what follows, but in our case, it is only because we humans find them easier to read and write. We will be thinking binary as we use them.

In other applications, there was a meaning to the base used in terms of the circuitry of the chip. In the early octal computers, storage locations were wired in parallel in multiples of 3, with groups of 3, 6, 24, and 48 binary digits being popular. In modern

parlance, we would have called them 3-bit, 6-bit, 24-bit, and 48-bit computers. Numbers stored on such computers had to be multiples of three binary digits long, and so octal was a convenient base to choose, notationwise.

In modern computers, storage positions are wired in multiples of 4, with groups of 4, 8, 16, 32, and 64 binary digits being popular. On these computers, hexadecimal is the convenient base, although 4 would also have worked.

Understand, however, that we are talking notation, not reality. Computers are binary. We choose to write numbers in base 8 or base 16 because that is more convenient for us humans.

By the way, the conversion method described above works just as well to the right of the base point as to the left. Consider 4.8_{16}. In base 2, that number is 100.1_2.

Storage and manipulation of integers

This article covers the implications of how computers store and manipulate numbers for numerical accuracy. So let me tell you that there are no such implications of the integer types. There is a one-to-one mapping of the integers of any base onto any other base, and that is sufficient to guarantee equivalency. With the integer types, you can think in any base you find appealing, even base 10.

Stata provides three integer types called `byte`, `int`, and `long` and being, respectively, 1, 2, and 4 bytes long. A byte is defined as 8 binary digits, so the numbers are 8, 16, and 32 binary digits long. Equivalently, a byte is two hexadecimal digits, and the numbers are 2, 4, and 8 hexadecimal digits long. These Stata types correspond to what modern computers use—that's why they were chosen—although modern 64-bit computers also provide a 64–binary digit integer that Stata does not make available to you. Its primary use is in recording memory addresses within the whoppingly large space that a 64-bit computer can provide.

You may have heard that some computers store digits left to right and others, right to left. That's true. A variety of names are used to label this schema. MSB and LSB are the most formal: they stand for most significant byte first and least significant byte first. In Stata, we often refer to these as HILO and LOHI computers. The system is also known as Big Endian and Little Endian, the word Endian coming from *Gulliver's Travels*, in which the Big Endian and Little Endian parties debated whether eggs should be opened at the big end or the little end.

Anyway, consider the integer 12345_{10}, which is 11000000111001_2 if we write from left to right. We could just as well adopt the habit of writing numbers from right to left: 10011100000011. Why would we want to do this? Think about how you add numbers by using the standard way of writing numbers from left to right: you have to start at the rightmost digits. Teaching grade-school children how to add and carry would be easier if we wrote numbers backwards. Computers add the same way we do; they just use a different base. Skipping all the way to the right requires a little extra circuitry and

a little extra time, and in the early days of microprocessors both were to be avoided. So computer engineers started storing numbers from right to left. That way, the least significant digit was right there, in the first position, where the computer needed it. By the way, the savings in circuitry and time occurs only with 8-bit computers. On wider computers, there is neither savings nor cost, one way or the other. In the early days of microprocessors, the big mainframes were not only bigger but also wider—another reason they got the name Big Endians. The Little Endians were the new, narrower microcomputers.

The manufacturers of the micros, however, were not consistent in their use of right to left when it came to notation. The left-to-right number 11000000111001_2 can be written in base 16 as 3039_{16}. So, if we write it from right to left, it makes sense that we should write 9303. The computer manufacturers, however, did not adopt that notation. They instead wrote 3930, which is an odd combination of right-to-left with left-to-right. It was, however, just a matter of notation. They reasoned like this:

Consider the number 11000000111001_2. The way they store the number in an MSB computer is 00110000 in the first storage location (a storage location holds 8 bits) and 00111001 in the second. Now, if I stored those bytes the other way around, I could save a little circuitry. I'll store 00111001 in the first location and 00110000 in the second. Within 8-bit storage locations, however, the wiring is in parallel, so there really is no meaning to order. I'm used to writing numbers from left to right, so within byte, I think I'll continue to write them that way. On paper.

The storage of negative integers is more clever, and all computer architectures agree on how this is done. The method is formally called "ones' complement plus one", but this is a case of the math being filled in after the fact and that the way computers handle negative numbers is that they don't. It's all a matter of interpretation.

This concept is most easily explained with analogy to a snooker table. On a snooker table, each player has a counter mounted on the side of the table. The counter has two digits and is base 10, of course. You can increment and decrement the counter. Snooker scores are positive but sometimes beginners are allowed to have negative values, at least for short periods. If you decrement a snooker counter that shows 00, it changes to 99. Decrement again, and it changes to 98. Increment it twice, and you are back to 00. That is nines' complement plus one notation. You just interpret 99 to be -1, 98 to be -2, etc., and you don't change the gearing on the counter at all.

That is what computer designers did. They just let the circuitry do what it wanted to do. Start with 00000000_2 and subtract 1, and circuitry that knows nothing about negative numbers will produce 11111111_2. Subtract 1 again, and you get 11111110_2. Add two and you are back at 00000000_2. So although you may read that a computer has 1-, 2-, and 4-byte integers, signed and unsigned (implying a total of six types), it in fact has only three. Whether a number is signed or unsigned is a matter of interpretation only. If a number is signed, then half the range is interpreted as being negative.

Storage and manipulation of real values

Modern computers store real values by using a 4- or 8-byte floating-point, binary standard. Every part of that has an implication for us programmers: the length (4 versus 8), the floating point, and the binary. First, let's focus on the floating-point part.

Modern, digital computers have no real understanding of real numbers. Reals are something more suitable for older analog computers. Digital computers can store only integers, and so the floating-point notation was developed. Since we are focusing on floating point and not the binary part, you can think in base 10 if you want to. Modern computers store real numbers, z, as a triple of integers (s, S, e) where the integers are given the interpretation

$$z = s \times S \times 10^e$$

In the above, s is called the sign and is either $+1$ or -1. S is the significand and is an unsigned integer with a fixed number of digits, say, five. e is a signed integer.

Seeing how this system works is fairly easy. Consider two real numbers, z_1 and z_2, represented by (s_1, S_1, e_1) and (s_2, S_2, e_2). Then

$$z_1 \times z_2 = (s_1 s_2) \times (S_1 S_2) \times 10^{e_1+e_2}$$

or, if you prefer, $(s_1, S_1, e_1) \times (s_2, S_2, e_2) = (s_1 s_2, S_1 S_2, e_1 + e_2)$. In this way, multiplication of reals is transformed into easy-to-perform integer calculations.

An implication of this implementation is that multiplication is fast and accurate, although not perfectly so. It is not perfectly accurate because S is stored with a fixed number of digits—we hypothesized five. When we multiply, we can lose digits, and the last digit we do record may be one off. If the computer performs calculations with guard digits (modern coprocessors do), we know that it will not be more than one-half off. The relative error is thus less than 0.5×10^{-5}.

Let's do an example. Let's calculate 4π. The number 4 in our notation is $(1, 40000, 0)$. Actually, it would be $(1, 40000, -4)$ in our notation as I defined it, but it is common to assume a base point (decimal point) between the first and subsequent digits of S, even on real computers, so we will do that. π is $(1, 31416, 0)$. The true value of 31416×4 (performed as an integer calculation) is 125664, but we can record only five digits, so we are left with 12566. The result is $(1, 12566, 1)$, or 12.566. Given the input, the relative error is $|12.566 - 12.5664|/12.5664 = 3.183 \times 10^{-5}$.

Division works the same way as multiplication.

As another example, obtaining square roots is easy, too. The $\sqrt{z}$ is either $(1, \sqrt{S}, e/2)$ if e is even or $(1, \sqrt{10S}/\sqrt{10}, (e-1)/2)$ otherwise. S we know is $0 \leq S < 10$, so we will need to write a square root subroutine that can produce square roots of values between 0 and 100, and only 0 to 100. That task should not prove too difficult. We could use Newton's method. Work it all out and you will discover that in the worst case, we lose only one digit. (In binary we need to be able to take square roots between 0 and 2—an even easier problem—and we still only lose up to one digit. Here the digit is binary.)

It is addition and subtraction that are the real killers, both computationwise and precisionwise. First, however, let's consider a case that does not cause precision problems. We wish to calculate $z_1 + z_2$, where $e_1 = e_2$ and $s_1 = s_2$. Then the decimal points line up, the signs are the same, and all we need to do is add S_1 and S_2. In doing that, we may have to discard a digit, but that is no worse than multiplication. Consider adding 1.2352 and 8.8231, that is, $(1, 12352, 0)$ and $(1, 88231, 0)$. $S_1 + S_2$ is 100583, but we can record only five digits, so we are left with $S = 10058$, and we must remember to increment the resulting e by one. The result is $(1, 10058, 1)$.

The greater the difference between e_1 and e_2, the more error we will have. Things can get so bad that adding $z1$ to $z2$ simply results in $z1$. Let's continue with $z1 = 1.2352$, and this time let's add $z_2 = .000044343$ to it. Our numbers are $(1, 12352, 0)$ and $(1, 44343, -5)$. The decimal points do not line up, so we denormalize the number with the smaller e until it matches the larger e value. So $(1, 44343, -5)$ can be written, with less accuracy, as $(1, 04434, -4)$, and then as $(1, 00443, -3)$, and then as $(1, 00044, -2)$, and then as $(1, 00004, -1)$, and finally as $(1, 00000, 0)$. That is the number we will add to z_1 (which has $e_1 = 0$), and the result will be, of course, $z1$.

Well that's just silly, you are probably saying to yourself. In real computer applications, with binary digits and lots of them, and whatever other refinements you might add, that doesn't happen, does it? In statistical applications, it does happen. It is common to calculate $\sum z_i$ over observations or, worse, $\sum z_i^2$. That is why, in linear regression, it is important that the routine remove the means and sum $(z_i - \bar{z})^2$. As $\bar{z}$ increases, accuracy falls. It is also why Stata does many calculations in quad precision. It is also why the mean-update formula for obtaining the sum is worth remembering:

> *Mean-update formula.* To calculate $S = \sum_{i=1}^{N} z_i$, calculate the mean, m (formula follows), and multiply by N. To calculate the mean, m, start with $m = 0$. Loop over the data and update m according to
>
> $$m = m + (z_i - m)/i$$
>
> This formula, reported in Knuth (1998, 232) where it was attributed to Welford (1962), is purely brilliant and based on two ideas. The first is to make the sum calculation based on the mean by simple multiplication, which cannot go wrong. The second is to calculate the mean by summing into something that is, at all times, approximately the magnitude of the mean. Even so, the formula is not a panacea. In our experience at StataCorp, the formula always yields better results than $\sum_i z_i$, but the formula never yields results as good as $\sum_i z_i$ calculated in quad precision, and only a little extra computer time is required to calculate the quad-precision result.

Another solution to the summation problem is to sort the data by $|z_i|$ and then sum the ordered values, from small to large. This approach is particularly useful when the z_i are not data but the terms of an iterative procedure. Many iterative approximations are written mathematically to produce the largest contributions first. Summing them in the

reverse order can sometimes improve accuracy markedly, especially if the approximation converges slowly.

Sometimes there is just no fix for the errors that can be introduced by addition and you must search for a different formula. The best example of this scenario is subtraction of nearly equal values of the same sign, or equivalently, addition of nearly equal values with opposite signs. Consider $z1 = 1$ and $z_2 = -.99999$. If the calculation is performed with guard digits, we will obtain .00001 as the answer. That answer has only one digit of accuracy. In integer form, that result is $(1, 10000, -5)$. The zeros following the 1 in S are just made up; no one knows what the digits really should be.

That is precisely what happened to us in the opening graph of Mills' ratio. We calculated `normalden(x)/(1-normal(x))`. You saw that we got an inaccurate result, but why? You know that it cannot be the division. Multiplication and division never introduce much error. The problem had to be with `1-normal(x)`. For large values of `x`, `normal(x)` approached 1, and the subtraction simply left too few digits of accuracy. To fix the problem, substitute `normal(-x)` because `1-normal(x)` = `normal(-x)`.

If you remember only one thing from this article, let it be never, ever to calculate $1 - \textit{something}$. Find another formula. The expression $1 - \textit{something}$ usually arises with distributions. When distributions are not symmetric, Stata (and Mata) provide separate functions for each tail. Use the appropriate function.

And finally, we come to the problem of bases. I said that modern computers store real numbers z as a triple of integers (s, S, e), where the integers are given the interpretation

$$z = s \times S \times 10^e$$

Of course, modern computers do not use base 10. They use base 2:

$$z = s \times S \times 2^e$$

Who would guess that such a minor change could introduce problems?

Actually, it does not introduce problems. Powers 2 or 10 are equally good, but different. No one has ever suggested that the universe prefers tenths. The problem is that you have a bias toward base 10 because of happenstance, and many of your favorite base-10 numbers have no exact binary representation. Remember when I said that there was a one-to-one mapping of the integers of any base onto any other base and that is sufficient to guarantee no implications? There is no one-to-one mapping across bases for floating-point numbers.

This statement should not surprise you. You know that there is no exact representation of 1/9 in base 10. In base 9, however, the exact representation is 0.1_9. In base 9, 1/3 is 0.3_9.

In base 2, there is no exact representation for .1, .2, .3, .4, .6, .7, .8, .9, .01, .02, and so on. In binary, 0.1_{10} is

$$.0001100110011001100110011001\ldots$$

and 0.01_{10} is

$$.0000001010001111010111010100011110101110\ldots$$

The repeating patterns are easier to see if we write the numbers in base 16:

$$\begin{aligned} .1_{10} &= 1.9999999999999\ldots_{16} \times 2^{-4} \\ .01_{10} &= 1.47ae147ae147ae\ldots_{16} \times 2^{-7} \end{aligned}$$

That lack of an exact representation means that financial values such as \$1.01 and \$4,502.20 are rounded the instant they are stored. Storing such numbers in `double`s rather than `float`s results in less rounding but will not eliminate it.

The error is not much. Rather than storing \$1.01, it may irritate you that the computer in fact stores 1.010000000000000009 (and even that is not precise; what is in fact stored is $1.028f5c28f5c29_{16}$), but the relative error is only 10^{-16}. If you store \$1.01 in float, what is stored is 1.009999990463256836 (actually, $1.028f5c_{16}$), and the (relative) error is now roughly 10^{-10}—still not much, but more bothersome. What is really bothersome, however, is how such numbers display when they are converted back to base 10.

If that bothers you, there is only one solution: store your dollar values in cents, that is, as integers. As I said, there is a one-to-one mapping of the integers of any base onto any other base, which is sufficient to guarantee no implications. Depending on size, store the number as a `long` or as a `double`. Yes, a `double`. It is true that `double` is a floating-point type, but it stores integers up to 9,007,199,254,740,992 precisely, and so the one-to-one mapping statement applies up to that value. Either way, you must deal with the rounding that already happened on input. Do that by typing

```
. generate double expense1_d = round(expense1*100)
```

Do not forget the `round()`, because otherwise you will have changed nothing. If `expense1` was 3,000.12 and stored as a float, what was stored was 3,000.1201171875 (i.e., $bb8.1ec_{16}$). Multiplication by 100 merely changes it to be 300,012.01171875.

Since you have gotten this far, let me give you the full definition of the IEEE Binary Floating-Point Arithmetic (IEEE 754) standard because so many of us use it and so few have actually read it:

> Real value z is stored as (s, S, e), where $z = s \times S \times 2^e$ and where
>
> s is $+1$ or -1;
>
> S contains 53 binary digits for `double` and 24 for `float`;
>
> $-1{,}023 \leq e \leq 1{,}023$ for `double` and $-127 \leq e \leq 127$ for `float`.

There are other details, but these are the important ones for us scientific programmers—and, here, the fact that the top power ($e = 1{,}023$ or 127) is reserved by Stata for the storage of missing values. The remaining facts have to do with storing IEEE infinities, NaNs ("not a number"), denormalized numbers, and how all of the above is packed into 8 or 4 bytes. For those interested, http://grouper.ieee.org/groups/754/ or "IEEE

floating-point standard" at http://en.wikipedia.org/wiki/IEEE_floating-point_standard is a good place to start.

Both Stata and Mata provide a numeric format that will let you see the (s, S, e) encoding: `%21x`, as Nick Cox pointed out in a recent tip (Cox 2006). For instance,

```
. display %21x 1.5
+1.8000000000000X+000
. display %21x 3
+1.8000000000000X+001
```

or in Mata,

```
: printf("%21x\n", -3)
-1.8000000000000X+001
```

Among other things, `%21x` makes it easy to spot numbers rounded to `float`:

```
. display %21x _pi
+1.921fb54442d18X+001
. display %21x float(_pi)
+1.921fb60000000X+001
```

For those interested in learning more about the substance of scientific, numerical programming, I recommend Knuth (1998).

1 References

Cox, N. J. 2006. Stata tip 33: Sweet sixteen: Hexadecimal formats and precision problems. *Stata Journal* 6: 282–283.

Knuth, D. E. 1998. *The Art of Computer Programming. Volume 2: Seminumerical Algorithms*. 3rd ed. Reading, MA: Addison–Wesley.

Welford, B. P. 1962. Note on a method for calculating corrected sums of squares and products. *Technometrics* 4: 419–420.

About the author

William Gould is President of StataCorp, head of development, and principal architect of Mata.